



# Quokka: A Fast and Accurate Binary Exporter

Alexis Challande, Robin David, Guénaél Renault

## ► To cite this version:

Alexis Challande, Robin David, Guénaél Renault. Quokka: A Fast and Accurate Binary Exporter. GreHack 2022 - 10th International Symposium on Research in Grey-Hat Hacking, Nov 2022, Grenoble, France. hal-03845728

**HAL Id: hal-03845728**

**<https://hal.science/hal-03845728>**

Submitted on 14 Nov 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Quokka: A Fast and Accurate Binary Exporter

Alexis Challande<sup>1,2</sup>   Robin David<sup>1</sup>   Guénaél Renault<sup>2,3</sup>

<sup>1</sup>Quarkslab {achallande, rdavid}@quarkslab.com

<sup>2</sup>Inria, École Polytechnique, Institut Polytechnique de Paris

<sup>3</sup>ANSSI guenael.renault@ssi.gouv.fr

## Abstract

Disassembling is the backbone for multiple workflows in binary analysis and offloaded to specialized tools. However, programmatically manipulating the dissembler’s results is cumbersome. Thus, we introduce QUOKKA, a binary exporter helping security practitioners to reuse the disassembler results in an *offline* context. QUOKKA is an open-source IDA plugin exporting a 20 MB program in less than 4 seconds to a compact binary format (7.4 MB). In this paper, we describe its inner workings and detail some of its usages.

## 1 Introduction

Analyzing binary programs often requires disassembling them. It is the backbone of security workflows for multiple topics like malware analysis, vulnerability research, or binary instrumentation. Thus, disassembling is crucial to analyze or securing untrusted or proprietary binaries whose source code is not available [4].

As correctly disassembling is an open problem [10], the security community has offloaded this task to specialized tools. Some are commercial like IDA [7], Binary Ninja [14], or Jeb [11], and others are open-source like Ghidra [9], BAP [2], or McSema [13]. The main problem faced by disassemblers is to recover information (e.g. symbols, types) lost during the compilation. Indeed, it is insufficient to convert a sequence of bytes into meaningful assembly instructions. A disassembler also needs to find references between code and data, recover functions boundaries, identify typical language structures (i.e. jumps or virtual tables), and reconstruct the Control Flow Graph.


While studying the inner workings of disassemblers is out of this work scope, we highlight that disassemblers are complex software that combines both algorithms (i.e. linear sweep, recursive descent) with correctness guarantees and heuristics with fewer guarantees to perform their tasks.

Disassembler	Exporter	Description
IDA	BinExport	Exporter from Zynamics
	Ghidra-IDA	Official Ghidra plugin to export a project from IDA to Ghidra
Ghidra	McSema	Exporter for McSema lifter
	BinExport	BinExport port for Ghidra
	Ghidra	Built-in exporter from Ghidra

Table 1: List of Different Binary Exporters

Usually heavy and complex softwares, disassemblers are inadequate to either perform custom analysis on a disassembled program, or to analyze multiple binaries at the same time. Indeed, a disassembler instance running in the background may use a few hundreds of megabytes in RAM effectively wasting resources if their functionalities are not used anymore. Moreover, their APIs may be convoluted and challenging to use. If only the disassembler’s *output* is needed for further analysis, it should be possible to extract this result to run *offline* queries using a binary export.

**Definition 1.** We define a **binary export** as a standalone file (e.g. usable without a disassembler) containing data from the disassembled binary.

In this paper, we present QUOKKA, a binary exporter for IDA. It exports most information recovered by IDA during the disassembly process and exposes them offline by generating a compact export file. QUOKKA is open-source and available on  GitHub<sup>1</sup>.

## 2 Existing Binary Exporters Review

We survey in Table 1 various existing binary exporters for the main disassemblers. The most common one is BinExport [3], a binary exporter created by Zynamics<sup>2</sup> and used in BinDiff [15]. BinExport supports three disassemblers backends: IDA, Ghidra and Binary Ninja and generates a binary file in the Protobuf format [6]. Because BinExport is tailored for BinDiff, it only exports information relevant for the diffing algorithms. Moreover, they do not offer bindings to allow a seamless usage of their exported file.

Ghidra provides an official exporter plugin for IDA. The plugin generates an XML file to extract some information from IDA and to import the project into Ghidra to continue the analysis. Because the objective is to reuse the result in another disassembler, the export does not contain any data regarding the instruction themselves. Moreover, the export file is quite large as XML is a textual format.

<sup>1</sup><https://github.com/quokka-project/quokka>

<sup>2</sup>Zynamics was acquired by Google in 2011.

Another solution to export the results of the disassembly is to use McSema. McSema is an executable lifter: it translates native machine code to LLVM IR. Its first step uses IDA to disassemble the target binary and generates a Protobuf with the information extracted from the disassembler. However, as the tool main objective is to generate LLVM bitcode, it uses a second tool to translate instructions. Thus, the first export does not contain information on the instructions themselves other than their addresses.

Instead of using complete solutions, academic works usually reimplement their extractor on top the disassembler API. For example, DeepBinDiff [5] uses a script to generate features vectors for each instruction and write them in a text file. This approach is not scalable as it creates huge and slow-to-parse files.

None of the existing exporters is adequate to be used as a general purpose exporter. They either lack bindings to seamlessly use the exported file, provide insufficiently optimized serialization files, or forget to export some crucial information. Thus, there exists a need to create for a new solution solving these problems.

### 3 Quokka: A Fast and Accurate Binary Exporter

QUOKKA’s objective is to be a generic binary exporter, suited for various contexts. It implements the following principles:

- Exhaustivity: To be used in various contexts, Quokka exports as much data as possible.
- Efficiency: To ease the integration inside analysis workflows and not create a bottleneck, Quokka is fast. The export time is negligible compared to the disassembly time.
- To avoid unnecessary disk usage and allow seamless export file sharing between users, Quokka export file is compact.

QUOKKA is composed of two independants parts:

- An IDA Plugin developed to address the limitations observed in other binary exporters that generates a compact export file.
- Python bindings enabling a seamless manipulation of the exported file.

In this section, we first list the exported items before briefly discussing its architecture, and detail some optimizations implemented in the plugin to reduce the exported file size *on the wire*.

#### 3.1 Exported Items

The features exported by QUOKKA are listed in Table 2 are being compared with both BinExport and Ghidra built in export. On a general point of view, QUOKKA’s export is more exhaustive than the two other tools: it exports every item exported by at least one of them.

		BinExport	Ghidra-XML	QUOKKA
Metadata	Name	✓	✓	✓
	Architecture	✓	✓	✓
	ISA	✓	✓	✓
	Compiler	✓	✓	✓
Layout	Segments	✓	✓	✓
	Code Layout	≈	✓	✓
Symbols	Name	✓	✓	✓
	Value	✓	✓	✓
	Type	✗	✓	✓
Data	Address	✓	✓	✓
	Type	✗	✓	✓
	Size	✗	✓	✓
	Name	✗	✓	✓
Graphs	Call Graph	✓	✗	✓
	CFG	✓	✗	✓
Comments	Address	✓	✓	✓
	Type	✓	✓	✓
	Content	✓	✓	✓
Functions	Name	✓	✓	✓
	Type	✓	✓	✓
	# Arguments	✗	✓	✓
Instruction	Mnemonic	✓	✗	✓
	Operand	✓	✗	✓
	Operand Type	✗	✗	✓
	Bytes	✓	✗	✓
	Address	✓	✗	✓
	Expressions	✓	✗	✓
	Xref (code, data)	✓	✗	✓
Basic Block	Address	✓	✗	✓
	Instructions	✓	✗	✓
	Type	✗	✗	✓
	Content	✓	✓	✓
Strings	Address	✓	✓	✓
	Content	✓	✓	✓
Data Structures	Structures	✗	✓	✓
	Enumeration	✗	✓	✓

Table 2: Comparison of Exporter Features

## 3.2 Quokka Architecture

QUOKKA's plugin is composed of about 3,500 C++ lines of code which targets IDA's last versions. The exporter works in three consecutive phases to generate a binary *Protobuf* file representing program exported data.

While QUOKKA can export every binary file loaded by IDA, it has only been tested with binaries respecting the following conditions:

- Architectures: X86, X64, ARM, Aarch64, Mips\*, and PPC\*
- Formats: PE, ELF, MacO, and DEX

However, it should be noted that the support for Mips and PPC is lackluster.

The first step is to consider elements outside the program address space. During this phase, the segments, the structures and the program metadata are exported. The second step is the main one. It performs a linear scan on the program address space and exports each element found during the memory scan. This phase is used to export the code and the data. Finally, the last phase is used to sort and resolve references between various items. This step is crucial as references are of the most important elements in the disassembler output.

## 3.3 Storage Optimization

QUOKKA aims to be compact to easily share the exports files and reduce storage utilization. To achieve smaller exported file, QUOKKA implements space optimization techniques adapted for the Protobuf format. We discuss below the main ones.

**Addresses and Offsets** Most program items (i.e functions or instructions) have an associated address within the program. As it is used in numerous workflows, it is an important data to export. However, programs usually have a large base address (e.g. 0x400000). In Protobuf, the size *on the wire* of an integer depends on its absolute value<sup>3</sup>, and it is more efficient to store relatively low integers.

Thus, in QUOKKA, function addresses are stored as offsets to the program base address, and block addresses as offsets to the function start. This optimization saves about 2 bytes per address. Moreover, instructions are stored without addresses, as this one is recomputed dynamically with their block starting address and the size of the previous instructions in the block.

**Data Deduplication** A program may use multiple times the same item, but at different addresses. For example, the instruction `push ebp` may be used in each function prologue. To improve the storage compactness, QUOKKA only stores items in a table and refer to them by their index in this table. To further improve compactness, deduplication tables items are sorted by usage frequency to have the most used items at the lower indexes.

---

<sup>3</sup>When using the `varint` encoding.

```

message Quokka {
    message Instruction {
        uint32 size = 1;
        uint32 mnemonic_index = 2;
        bool is_thumb = 4;
    }

    repeated string mnemonics = 8;
}

```

Listing 1: Extract of QUOKKA schema definition

**Defaults Values** In Protobuf, each field has a type. As an optimization, Protobuf does not write on the wire a field value when it is the type default value. QUOKKA leverages this optimization, notably for boolean. For example, each instruction is stored with the `is_thumb` boolean (to allow the export of ARM Thumb instructions). In most cases, the field will be `False`, which is the boolean default value in Protobuf.

### Example

Listing 1 shows an extract of QUOKKA’s schema definition. It illustrates the optimizations previously mentioned:

- The instruction message has no address.
- The instruction’s mnemonic is stored in a deduplication table and only its index is used.
- The field `is_thumb` is only written for thumbs instruction in ARM binaries because the default value for boolean is `False` in Protobuf.

## 4 Using Quokka

QUOKKA is a two-fold tool. Its first part is the IDA plugin already presented in the previous sections. The second part is QUOKKA’s Python bindings. They offer an effortless way to manipulate the exported file. In this section, we will first explain how to generate an export file, before demonstrating some of the binding capabilities.

The examples in this section are limited and additional documentation is available with the project.

### 4.1 Exporting a Binary File

Using QUOKKA from the command line is straightforward as exposed in Listing 2. When using IDA graphical interface, the plugin also registers a shortcut (by default `Alt+A`) to generate an export file.

```
$ idat64 -OQuokkaAuto:true \
-OQuokkaLog:Info \
-OQuokkaFile:docs/samples/qb-crackme.qk \
-A docs/sample/qb-crackme
```

Listing 2: Using QUOKKA directly from IDA command line

```
import quokka

program = quokka.Program("docs/samples/qb-crackme.qk", "docs/sample/qb-crackme")
for function in program:
    print(f"Function {function.name} starts at 0x{function.start:x}")

    if function.name == "main":
        for instruction in function.instructions:
            print(instruction.mnemonic)
```

Listing 3: Listing all functions in a Program with QUOKKA

## 4.2 Manipulating the Exported File

Due to the various optimizations, QUOKKA’s Protobuf file is challenging to use directly. Thus, we provide Python bindings to allow for an easy manipulation of the program. While generating an export file requires to have IDA installed, it is not needed to load and use the exported file.

The code in Listing 3 loads an exported file and uses it to list every functions inside the program. It also prints the mnemonic of instructions in the main function. This simple example is not useful *per se* but demonstrates some of QUOKKA functionalities.

## 5 Evaluation

### 5.1 Dataset

To compare QUOKKA with BinExport<sup>4</sup>, we select typical binaries present on our systems at the time of the writing with properties listed below:

- We consider multiple architectures as the exporter should be architecture agnostic.
- We want to consider various binary file formats to test their support.

As the export time and result size is a factor of the initial binary size, we select binaries from 25 kB and up to 46 MB. In Figure 1, we illustrate the binary sizes of the samples we use in the next experiments.



Binary Name	Arch.	Format	Binary size
MachO-OSX-x86-ls	x86	MachO	34.86 kB
pe-Windows-x86-cmd	x86	PE	294.50 kB
elf-Linux-x86-bash	x86	ELF	792.14 kB
elf-Linux-lib-x86.so	x86	ELF	1.08 MB
delta_generator	x86	ELF	16.49 MB
wpa_suppl	x86	ELF	21.64 MB
MachO-OSX-x64-ls	x86_64	MachO	38.66 kB
pe-Windows-x64-cmd	x86_64	PE	337.00 kB
x64_delta_generator	x86_64	ELF	15.28 kB
elf-Linux-x64-bash	x86_64	ELF	904.82 kB
elf-Linux-lib-x64.so	x86_64	ELF	1.09 MB
ctags	x86_64	ELF	4.59 MB
ts3server	x86_64	ELF	7.73 MB
mdbook	x86_64	ELF	10.67 MB
llvm-opt	x86_64	ELF	33.83 MB
clang-check	x86_64	ELF	46.83 MB
crackmap	MIPS-32	ELF	25.54 kB
busybox-mips	MIPS-32	ELF	352.48 kB
elf-Linux-Mips4-bash	MIPS-32	ELF	882.38 kB
HelloWorld-MachO-2	armv7	MachO	89.64 kB
HelloWorld-MachO	armv7, armv8	MachO	299.06 kB
elf-Linux-ARMv7-ls	armv7	ELF	88.68 kB
elf-Linux-ARM64-bash	armv8	ELF	827.54 kB
busybox-powerpc	PPC-32	ELF	1.10 MB
dex38.dex	-	DEX	11.48 kB
classes.dex	-	DEX	3.53 MB

Table 3: Datasets Binaries

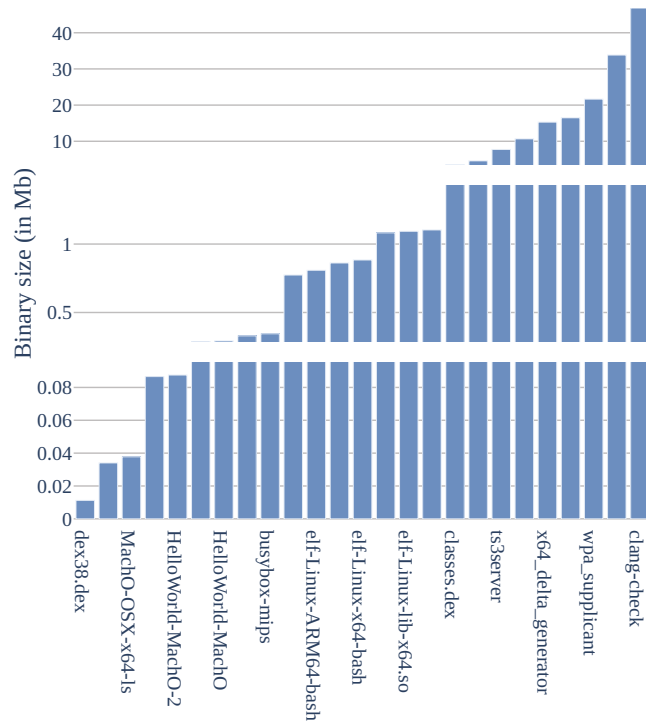


Figure 1: Samples Sizes

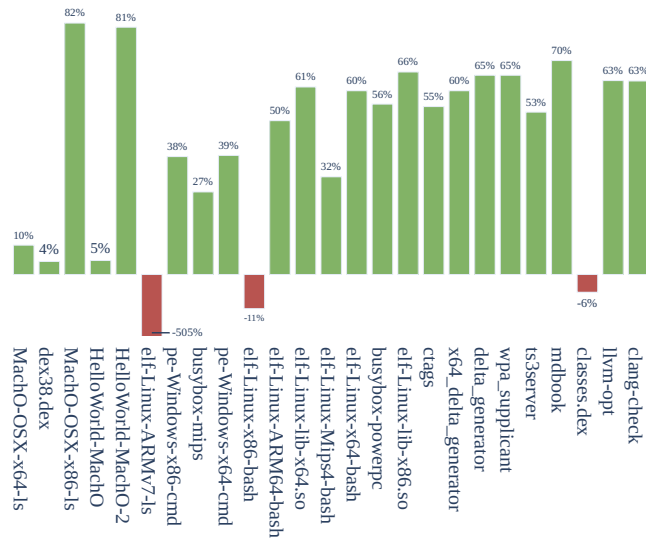


Figure 2: Duration: QUOKKA vs BinExport

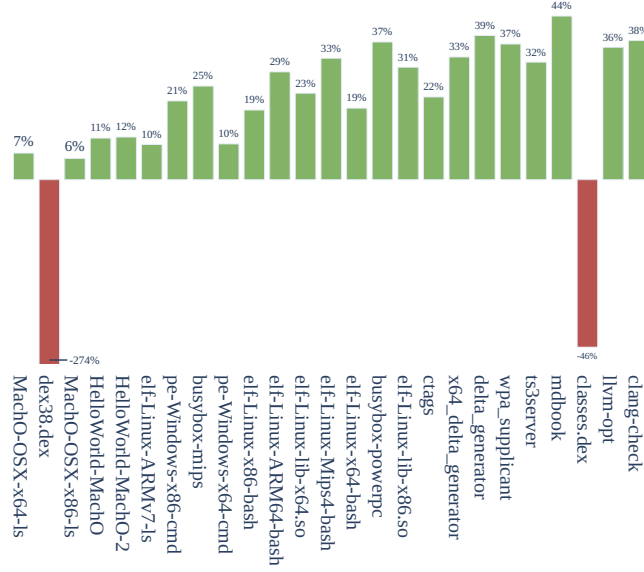


Figure 3: Size: QUOKKA vs BinExport

## 5.2 Efficiency

We first assess QUOKKA’s efficiency by looking at the difference regarding the export duration between BinExport and QBinExport for our samples. First, it should be noted that the two tools are fast: BinExport needs 11s to exports a 20 MB program (wpa\_suppllicant) while QUOKKA requires 3s. In comparison, IDA takes 44s to generate the database for the same program.

However, while exporting more features, QUOKKA’s optimizations are successful as the exporter is faster with a 63% improvement for the largest binary and a median 54% improvement for the dataset. Figure 2 displays the improvement percentage between QUOKKA and BinExport.

The extensive comparison of the exported files for `elf-Linux-ARMv7-ls` did not yield to any conclusive explanation in regards to the duration difference.

## 5.3 Compactness

IDA analysis results are stored in its database, a proprietary format based on B-Trees. It is possible to share this file across users but it is not very compact: IDA’s database for a 22 MB binary weights 51 M and the median size increase factor in our dataset is 12.

We also assess QUOKKA’s export compactness. As previously stated, QUOKKA exports strictly more data than BinExport. However, thanks to the optimization on size presented in the last section, QUOKKA manages to be more compact for most binaries. Indeed, the median improvement is 22%.

<sup>4</sup>Version 12

```

def extract_features(function: quokka.Function):
    vector = [
        # In / Out degrees of the function
        (function.in_degree, function.out_degree),
        # Function bytes
        function.bytes,
        # Functions used
        set(imp.name for imp in function.calls if imp.type ==
            ↪ FunctionType.IMPORTED),
        # Function size
        function.end - function.start,
        # Number of basic blocks
        len(function.graph),
        # Bag of mnemonics
        set(inst.mnemonic for inst in function.instructions),
    ]
    return vector

```

Listing 4: Extracting some features from a function

```

def candidate_functions(program: quokka.Program):
    for function in program:
        # Filter functions with less than 10 basic blocks
        if len(function.graph) <= 10:
            continue

        if "malloc" in function.calls and "free" not in function.calls:
            print(f"Function {function.name} calls malloc without free")

```

Listing 5: Searching for interesting functions

QUOKKA exported files are smaller for each sample except the two DEX files. There are two explanations for this. Firstly, QUOKKA exports the layout (i.e. if a region is used for code or data) which changes often for DEX, requiring multiple objects. Secondly, QUOKKA also exports data structure which takes up to 4.8 MB in `classes.dex`, thus explaining the 5 MB difference between the two formats.

## 6 Potential Usages

Exporting as many data as possible from a binary is interesting as a backbone for other applications. In this section, we list some potential usages for QUOKKA.

**Feature Extraction** Most machine learning approaches for binary analysis require to generate features from the disassembly code. For example,  $\alpha$ Diff’s authors [8] developed a custom plugin to extract data from the function in the binary (function bytes, in and out-degrees, and the imported function sets). All these informations are directly available in QUOKKA and the Listing 4 shows how to extract them (and some others). Using a battle-tested plugin allow researchers to focus on the novelty of their research instead of wasting valuable time with IDA API.

```

def hash_func(function: quokka.Function):
    local_hash = hashlib.sha256()
    for instruction in function.instructions:
        local_hash.update(instruction.bytes)

    return local_hash

prog1 = quokka.Program("prog1.qk", "prog1")
prog2 = quokka.Program("prog2.qk", "prog2")

for func_name in set(prog1.fun_names).intersection(prog2.fun_names):
    func1 = prog1.fun_names[func_name]
    func2 = prog2.fun_names[func_name]

    if hash_func(func1) != hash_func(func2):
        print(f"Function {func_name} has changed between prog1 and prog2")

```

Listing 6: Finding changed functions between two program

**Binary Analysis** QUOKKA allows one to easily filter functions by user-defined criterias. For example, the Listing 5 shows how to search for large functions calling `malloc` without calling `free`. If the listing could be extended to list every function using *unsafe* functions, performing a real dataflow analysis would require to use another tool (i.e. Triton [12], or BinCAT [1]).

**Side by Side Analysis** QUOKKA is not limited at loading a single binary in memory. Thus, it can be used to compare side-by-side two binaries at the same time. This could prove useful when analyzing two versions of the same program and finding which functions have changed. Listing 6 demonstrates this usage.

This section only scratched the surface of what is possible to do with QUOKKA. However, it is only a library enabling other tasks and workflow, and not a tool used at the end of an analysis pipeline.

## 7 Future Work

While thorough, QUOKKA still lacks the export of some features listed below.

- **Type information:** IDA allows users to define types and to apply type information onto the disassembly. This information helps the reverser understanding the data flow inside the binary and can be used for other tools to perform analyses (pointer analysis, liveness analysis). Thus, it could be valuable to expose them outside the disassembler.
- **Decompilation output:** IDA's SDK offers an API to manipulate the decompilation output, i.e. a C code generated from the disassembly. Exporting the decompiled code would broaden the usage possibilities of QUOKKA.


As IDA already offers API functions to manipulate and query such data, exporting these elements would only require further engineering work. However, working with the disassembler API is time-consuming.

Currently, QUOKKA is an IDA plugin. Another improvement for the tool would be to also accept other backends (such as Ghidra or Binary Ninja). Thus, it could act as an IR where an analyst query workflow could be written once, and the backend modified depending on the disassembly selected.

## 8 Conclusion

QUOKKA is a fast and complete binary exporter for IDA Pro. It timely generates compact binary file containing most IDA intelligence and allows reusing the result in an offline setting. QUOKKA has notably already been used for various tasks in Quarkslab for 2 years.

The export generated by QUOKKA can be used in various workflows such as features extraction for machine learning, vulnerability research, or binary diffing. It can also be used to work directly on the disassembly without needing a disassembler instance.

The tool is open-source and available online on  GitHub<sup>5</sup> and can be used for various security workflows.

## References

- [1] Biondi, Philippe et al. “BinCAT: Purrfecting Binary Static Analysis”. In: *Symposium Sur La Sécurité Des Technologies de l’Information et Des Communications*. 2017. Rennes, France, June 2017. URL: [https://www.sstic.org/2017/presentation/bincat\\_purrfecting\\_binary\\_static\\_analysis/](https://www.sstic.org/2017/presentation/bincat_purrfecting_binary_static_analysis/).
- [2] David Brumley et al. “BAP: A Binary Analysis Platform”. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, July 2011, pp. 463–469. ISBN: 978-3-642-22110-1. DOI: [10.1007/978-3-642-22110-1\\_37](https://doi.org/10.1007/978-3-642-22110-1_37).
- [3] Christian Blichmann. *BinExport*. Google. Nov. 2021. URL: <https://github.com/google/binexport> (visited on 06/03/2022).
- [4] Dennis Andriesse and Xi Chen. “An In-Depth Analysis of Disassembly on Full-Scale X86/X64 Binaries”. In: (Aug. 2016), p. 19. ISSN: 978-1-931971-32-4.
- [5] Yue Duan et al. “DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing”. In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, Feb. 2020. ISBN: 978-1-891562-61-7. DOI: [10.14722/ndss.2020.24311](https://doi.org/10.14722/ndss.2020.24311).
- [6] Google. *Protocol Buffers - Google’s Data Interchange Format*. Protocol Buffers. May 2022. URL: <https://github.com/protocolbuffers/protobuf> (visited on 06/03/2022).

---

<sup>5</sup><https://github.com/quokka-project/quokka>

- [7] Hex-Rays. *IDA Pro - Interactive Disassembler*. Hex-Rays. May 2022. URL: [www.hex-rays.com/idapro/](http://www.hex-rays.com/idapro/) (visited on 08/01/2021).
- [8] Bingchang Liu et al. “ $\alpha$ Diff: Cross-Version Binary Code Similarity Detection with DNN”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. Montpellier, France: ACM Press, Sept. 2018, pp. 667–678. ISBN: 978-1-4503-5937-5. DOI: [10.1145/3238147.3238199](https://doi.org/10.1145/3238147.3238199).
- [9] NSA. *Ghidra Software Reverse Engineering Framework*. National Security Agency. May 2022. URL: <https://github.com/NationalSecurityAgency/ghidra> (visited on 06/03/2022).
- [10] Chengbin Pang et al. “SoK: All You Ever Wanted to Know About X86/X64 Binary Disassembly But Were Afraid to Ask”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2021, pp. 833–851. ISBN: 978-1-72818-934-5. DOI: [10.1109/SP40001.2021.00012](https://doi.org/10.1109/SP40001.2021.00012).
- [11] PNF Software. *JEB Decompiler by PNF Software*. PNF Software. May 2022. URL: <https://www.pnfsoftware.com/> (visited on 06/03/2022).
- [12] Jonathan Salwan. “Triton : Framework d’exécution concolique et d’analyses en runtime”. In: *Symposium sur la Sécurité des Technologies de l’Information et des Communications*. June 2015, p. 25.
- [13] Trail of Bits. *McSema*. Trail of Bits. Apr. 2021. URL: <https://github.com/lifting-bits/mcsema> (visited on 06/03/2022).
- [14] Vector 35. *Binary Ninja*. Vector 35. May 2022. URL: <https://binary.ninja>.
- [15] Zynamics. *BinDiff*. Zynamics/Google. June 2021. URL: <https://www.zynamics.com/bindiff.html>.