Tools and Models for Software Reverse Engineering Research

Thomas Faingnaert*
thomas.faingnaert@ugent.be
Ghent University
Ghent, Belgium

Tab Zhang*
tab.zhang@ugent.be
Ghent University
Ghent, Belgium

Willem Van Iseghem
willem.vaniseghem@ugent.be
Ghent University
Ghent, Belgium

Gertjan Everaert
gertjan.everaert@ugent.be
Ghent University
Ghent, Belgium

Bart Coppens
bart.coppens@ugent.be
Ghent University
Ghent, Belgium

Christian Collberg
collberg@cs.arizona.edu
The University of Arizona
Tucson, AZ, USA

Bjorn De Sutter
bjorn.desutter@ugent.be
Ghent University
Ghent, Belgium

Abstract

Software protection researchers often struggle with the evaluation of MATE software protections and attacks. Evaluations often are incomplete and not representative of the practice. This can in part be explained by a lack of standardized, generally applicable models, tools, and methodologies for evaluating how reverse engineering attack strategies are executed. The framework of related components proposed in this paper is an attempt to provide exactly that. It includes a meta-model and supporting tools for modeling the knowledge that reverse engineers acquire as they execute their strategies, a meta-model to estimate the required effort of those strategies, and tools to capture strategic activities from data streams collected during human reverse engineering experiments. Their use is demonstrated on three example reverse engineering strategies.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

Keywords

reverse engineering; strategy modeling, simulation, and capturing

ACM Reference Format:
Thomas Faingnaert, Tab Zhang, Willem Van Iseghem, Gertjan Everaert, Bart Coppens, Christian Collberg, and Bjorn De Sutter. 2024. Tools and Models for Software Reverse Engineering Research. In *Proceedings of the 2024 Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks (CheckMATE '24), October 14–18, 2024, Salt Lake City, UT, USA.*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3689934.3690817

*Faingnaert and Zhang share dual first authorship.

CheckMATE '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-1230-2/24/10
https://doi.org/10.1145/3689934.3690817

1 Introduction

A recently published survey on evaluation methodologies for obfuscated software identified major issues in that research domain [40]. The authors observed that few papers on the generation, deobfuscation, and analysis of obfuscated software evaluate how obfuscations resist real-world attack scenarios. They also observed little (re)use of analysis and deobfuscation tools, in particular of state-of-the-art tools and of the flexibility and extensibility such tools offer. They also observed that few papers evaluate potency, resilience, and stealth of software protections (SPs) being targeted by man-at-the-end (MATE) attacks. Given how hard it is to use tools in the way real-world attackers use them, these observations regarding important threats to validity of research results do not surprise. To the contrary, the survey confirmed the gut feeling of many participants to the 2019 Dagstuhl Seminar on Software Protection Decision Support and Evaluation Methodologies [13] regarding the poor state of evaluation methodologies in SP research. The survey authors then formulated a call to arms for the development of a toolbox of easily reusable software analysis tools with which real-world attacks can be mimicked. In this paper, we present such a toolbox.

The survey authors also found that there exists no standard, general evaluation methodology for SPs. Given the diverse objectives that SPs can serve, including the wide range of attack strategies they need to mitigate, the lack of a completely standardized evaluation methodology for all forms of MATE SP is understandable. That should not imply, however, that we should abandon all effort to standardize evaluation methodologies. Developing high-level evaluation methodologies for certain classes of attack strategies, if possible, could still enable more reuse of evaluation tools and make research results more comparable, thus improving the productivity of MATE SP researchers and helping them to mitigate threats to validity. In this paper, we present a skeleton for such a methodology.

The research presented here is a response to that call to arms. To help the SP research community overcome the mentioned issues, we decided to design and implement a framework for modeling reverse engineering (REing) activities, with an initial focus on localization and comprehension activities commonly used in MATE attack strategies [7, 38]. Our framework consists of three components:

(1) A *meta-model of the knowledge* obtained by reverse engineers (REs) while executing their strategies, and tool support for the instantiation of concrete models of that meta-model by means of a broad range of existing, state-of-the-art REing tools. This enables researchers to simulate real-world attacks.
(2) A *meta-model of the effort* required to execute those strategies, and methods and tools to estimate that effort on concrete models. This enables researchers to estimate the effort that real-world attackers might have to invest, and hence to estimate the impact that SPs might have.
(3) *Strategy capturing tools* for (semi)automatically capturing REing activities from recordings of humans tackling REing challenges. This enables researchers to identify which navigation methods REs use in search of artifacts, and which artifacts they visit as they try to comprehend code. This can help researchers validate the relevance and representativeness of the attack strategies and models used in their research. It can also help them to identify new relevant attack strategies they were previously unaware of, thus positioning them to design, instantiate, and use more accurate models.

The contributions of this paper are threefold: (1) the presentation of the framework and component designs; (2) their demonstration on three example attacks: cryptographic key extraction, license key check localization, and game cheat opportunity localization; (3) the open-sourcing of all our proof-of-concept tool support.

Section 2 introduces the example attacks strategies. Section 3 presents the knowledge meta-model and tool support for instantiating concrete models, and demonstrates it on the examples. Section 4 discusses the effort meta-model and how it applies to the examples. Section 5 discusses our tools for extracting code localization and comprehension activities in human experiments, explaining their capabilities on one of the examples. Section 6 discusses related work, and Section 7 draws conclusions and looks forward.

2 Example Reverse Engineering Strategies

Throughout this paper, we will rely on three example localization strategies to illustrate the constructs, models, and methods of our framework. In these strategies, REs rely on domain knowledge to identify intermediate artifacts first, which lead them towards their final goal. We call such intermediate and final artifacts mileposts.

2.1 Dynamic Cryptographic Key Extraction

Scenario. On some scenarios a program needs to encrypt or decrypt data with a (secret) key that is embedded or computed (e.g., with a key derivation function) in the program. A MATE adversary can then try to extract the key from the program.

Strategy. This is a dynamic REing strategy: most information is collected from execution traces, some is obtained with a debugger. In fact, this attack strategy is an extension of, and improvement over the K-Hunt strategy for automated key extraction [25].

Mileposts. This strategy involves four mileposts:

(a) *Basic blocks likely performing crypto* are identified in traces based on their mix of operations, their execution counts when crypto is enabled/disabled, how those counts scale with varying input sizes, and their operands' values' entropy.
(b) *Basic blocks likely loading/storing the data consumed/produced in the crypto operations.* These are identified by following the data dependencies starting from the previous milepost.
(c) *Operands of instructions in those basic blocks likely corresponding to key operands.* These are identified by taint-tracking where the operands originate from, by their instruction types, by the variations in their values within and over multiple runs, as well as the likelihood of their values being non-key values, and by the sizes of the buffers they refer to.
(d) *Key operand values.* These are identified by logging, at debugger breakpoints, the values of the identified key operands in the order in which they occur during the execution.

Defenses. This attack is not easily mitigated with obfuscations, because the features exploited in the different steps to prune the search spaces of the subsequent mileposts are so fundamental. While some obfuscations (e.g., anti-taint protections, constant-time transformations, code duplication) can hamper some of the pruning/selection heuristics, an ablation study revealed that the attack strategy is robust enough to tolerate the failure or lowered precision of a limited number of heuristics. In other words, mitigating the attack requires combining multiple expensive SPs, resulting in huge performance overheads, and even then the impact is still minimal, in the sense that the number of candidate code fragments found for each milepost only grows minimally.

2.2 License Key Checks Localization

Scenario. The second strategy targets naive implementations of software license checkers that (i) take a license key as input; (ii) extract subkeys and optionally perform arithmetic on them at various program points; (iii) perform checks on them at various program points and update the state of the license checker to "failed" upon checks failing; (iv) check that state at various program points and react in case the state is "failed" by outputting an error message and halting execution. To learn how to generate license keys themselves, MATE adversaries can try to identify the performed checks.

Strategy. This is a mostly static strategy using capabilities of interactive disassemblers such as IDA, Ghidra, and Binary Ninja. As such, it nicely complements the previous strategy.

Mileposts. This strategy involves six mileposts:

(e) *Error strings embedded in the program* are identified by extracting all strings from the binary and filtering for the most interesting ones. Potential options for this filtering are to execute the program with invalid keys and observe the exact error message, or to sort the strings, giving a higher weight to words such as "invalid", "activation", "code", "serial", "key", "wrong", "authorization", "incorrect", etc.
(f) *Code referencing the error strings.* To identify these fragments, the RE relies on interactive disassemblers to list all instructions referencing a global data element such as a string.
(g) *Reaction code fragments* output the error strings. They can be identified by browsing through the disassembled code, following "links" in the form of data dependencies that the disassemblers have extracted from the disassembled code or that dynamic analysis have recorded in execution traces.

(h) *Reaction triggers* are conditional control transfers that decide whether a reaction fragment gets executed. They can be identified by browsing through the disassembled code, following "links" in the form of control dependencies when those are available from automatic analyses, or by browsing manually through control flow graphs (CFGs).
(i) *Subkey checks.* These computations produce the trigger predicates. They are found in their program slices, by following data dependencies, or by browsing through the CFGs.
(j) *Subkey extraction from key and computations.* In a similar manner, these computations that are performed on the inputs to the key checks can be identified. Indeed, subkey checks and subkey extraction and computation code form the backward program slice of the reaction triggers.

Defenses. This strategy is easily mitigated. To prevent the first step, it suffices to replace the strings embedded in the program binary with code that regenerates the strings on the fly, with a so-called static-to-procedural conversion [27]. Dynamic analyses are then needed to identify the third milepost. A complementary defense is to insert decoys at various program points: code fragments that resemble the true mileposts, and that seemingly relate to each other and to the real mileposts in ways that make them hard to differentiate from real mileposts. For example, fake subkey checks can be sprinkled throughout the application that alter the data that encodes the license manager state but that do not alter the encoded state, fake references to error strings can be included, intermediary computations can be inserted to complicate the data dependencies between the various mileposts, etc. All of them will result in more candidates being labeled as potential mileposts, and hence more code fragments will have to be visited and studied by the RE to differentiate real mileposts from decoys.

2.3 Game Resource Hack Localization

Scenario. With game resource hacks, cheaters wish to modify the amount of resources (health, currency, ammunition, ...) of their character in the game [6]. To achieve this, one can patch the code fragments in the game that update the resources.

Strategy. This strategy to identify the resource's location complements the previous two ones by building on a completely different type of analysis technique, namely interactive memory scanning.

Mileposts. This strategy comprises two mileposts:

(k) *Resource data location.* Tools such as CheatEngine [8] or scanmem [37] can be used to identify the memory locations in a running game's address space that hold the resource data. The cheater then plays the game up to some point, halts the process, and scans its memory space for values corresponding to their amount of resources as shown on screen. Such a scan will typically yield multiple locations that store the searched values. Most of them will happen to store the same bytes by coincidence. To prune those, the attacker will play some more, such that the relevant values change, and repeat the scan, narrowing the search space to those locations now containing the changed values. This narrowing can be repeated multiple times until only a single location remains. The number of required iterations is typically not known a priori and can actually vary from run to run.
(l) *Resource hack location.* After attaching a debugger to the still running game and setting a watchpoint on the found resource data location, the code fragments accessing and/or updating that data can easily be found.

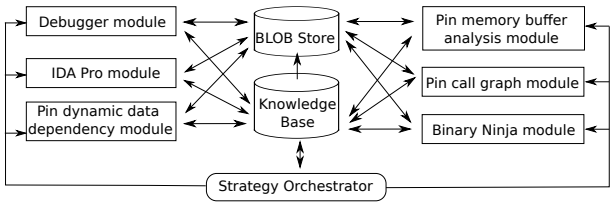


Figure 1: Overview of the REing KB, enrichment modules, a strategy orchestrator, and a BLOB store.

Defenses. Obfuscations can easily make the cheater require many more memory scan iterations and more complex pattern recognition heuristics. For example, the game could store the resources in non-standard number encodings by means of residue number coding [14]. REs then initially do not know how displayed numbers map onto stored ones, so they will need more complex heuristics to identify and prune the locations that store the searched values.

While modern commercial games include anti-tamper protections to prevent in-place code patches, we still think this REing strategy is relevant, if only because the first milepost is also relevant for more advanced cheats, such as attacks aiming to build a *pointer chain*, i.e., a sequence of fixed-offset dereferencing operations starting at a static or stack root address, for use in out-of-process cheats.

3 Reverse Engineering Knowledge Meta-Model

Our framework's first component enables the modeling of all knowledge that REs (and their tools) collect on their target programs and exploit while conducting REing activities. Concretely, we want to store that complex, structured information in a knowledge base (KB) with which enrichment modules (EMs) can then interact to extract and/or inject information. These EMs can do so by reasoning about the already collected information, but also by invoking analysis tools from the RE's toolbox, as shown in Figure 1. Strategy orchestrators (SOs) control the invocation of EMs, mimicking REs that invoke tools to collect knowledge as they execute strategies.

The KB then in effect stores a model of the attacker's knowledge on the targeted app. A meta-model (similarly to a grammar for a programming language) describes the allowed structure of such a model, formally defining its syntax and rules. In this section, we propose such a meta-model, and we present a set of EMs to instantiate concrete models and to interact with them.

3.1 Requirements

The knowledge that REs collect and on which they reason, spans a wide range of concepts. This is already evident from the above example attack strategies. Beyond those examples, various works inspired us on how to model that knowledge. The survey from Schrittwieser et al. on obfuscation vs. program analysis [38] provides an extensive overview of how different types of program

analysis, and hence different types of analysis results, are used by REs to defeat obfuscations. The taxonomy of Ceccato et al. of REing concepts is an additional important source of inspiration [7]. Ceccato et al. extracted REing concepts from reports by professional pen testers and interviews with an amateur using systematic qualitative analysis. This taxonomy includes 169 concepts and contains four models of hacker activities which relate to (i) obtaining code comprehension, (ii) making and testing hypotheses and building and executing attack strategies, (iii) choosing, customizing, and creating new tools, and (iv) defeating SPs. The recent survey by De Sutter et al. analyzes how SP publications deploy analysis techniques and concrete tools to evaluate the strength of obfuscations [40]. Beyond obfuscations, the practice of REing was studied in many papers (e.g., [26, 39]), with which we are also familiar.

Our knowledge meta-model aims to cover all constructs and models that are considered analysis results in those publications, as well as the ones we know from our experience. We hence first inferred some requirements and opportunities.

First, REs consider static artifacts at many levels of abstraction, all of which are represented with graphs in the programming language design and implementation domain, such as CFGs, abstract syntax trees, intermediate representation trees, and call graphs. Furthermore, software engineers as well as REs design and think of programs as a hierarchy comprising libraries, functions, basic blocks, and instructions. The composition of code artifacts is hence naturally modeled as a tree. There are exceptions, however. For example, the Binary Ninja disassembler can assign basic blocks to multiple functions when they are reachable from multiple function entry points through intra-procedural control transfer idioms [46].

Secondly, we identified four types of information that REs collect:

(1) *Artifacts* existing at various levels of abstraction: instructions, buffers, pointers, data structures with certain shapes, etc. that make up the static program or its dynamic state;
(2) *Relations* between artifacts, such as data and control flow dependencies between code artifacts, which operations access which data structures, etc.;
(3) *Properties* of artifacts and relations, such as their size, them being tainted or not, statistical information about their occurrences in a program execution (e.g., a trace), etc.;
(4) *Mappings* between concrete and more abstract artifacts, such as bytes forming a key, instructions forming a key check, and shapes on the heap forming abstract data structures.

All of these results can easily be modeled with graphs: artifacts at various levels of abstraction correspond to nodes, edges model relations or mappings between the artifacts, and properties can be modeled by labeling nodes and edges with them.

Third, the range of concrete types of artifacts, relations, properties, and mappings that need to be modeled is vast, and not limited to a predetermined set. For example, REs do not stop at identifying a generic "data structure in memory". Instead, once they discover that some data structure is a linked list, they will start to reason about that data and the operations thereon at the algorithmic level. They will label the artifacts as such, either in their mental model or in their tools, e.g. by renaming symbols such as `DAT_1234` in a disassembled binary to `linked_list0x1234` and the anonymous function `FUN_fedcba` to `linked_list_insert`. For our meta-model to cover all REing activities, it needs to support any concrete or abstract, domain-specific or general-purpose, basic or expert-level data structures and algorithms that REs might ever care about. In short, the types of nodes, edges, and properties need to be *flexible*.

3.2 Design

The above leads us to use graphs to model the knowledge that REs collect. The proposed meta-model hence defines what the graphs that instantiate this meta-model can look like.

First, nodes have *types*, can be identified by their unique *identifier*, and have a *set of properties*. Types, identifiers, and properties are not limited to predetermined sets of names or values. Each property can be a single value, but also a key-value store. This allows us to model related properties, such as the execution counts of instructions observed for different inputs. Similarly to nodes, (directed and undirected) edges also have a type, an identifier, and properties, none of which are limited to predetermined sets. The type of an edge corresponds to the type of relation between their source and sink nodes. Since nodes can be related in multiple ways, multiple edges can connect a pair of nodes, resulting in multigraphs.

3.3 Implementation

3.3.1 Knowledge Base. We store the graph model collected on a target app in a KB, we use a graph database (DB) [30] that is optimized to store graph data and to query relations and patterns of relations. Concretely, we use Neo4j [29]. In Neo4j, nodes and edges have labels in which we can store their type. Furthermore, nodes and edges in the DB have properties, which we can use to store the properties of all artifacts on which a RE collects information. In our current implementation, we implement key-value-store properties by using the keys as prefixes or suffixes in property names.

3.3.2 Binary Large Objects. Some tools produce massive outputs that are essentially arrays of bytes, such as execution traces and memory dumps. DBs are not designed for storing massive arrays internally. Instead, the best practice is to store them externally in *BLOBs* or Binary Large OBjects, and to store URLs to them in the DB. We follow that best practice, hence the BLOB store in Figure 1. Also the program binaries themselves, and input files to them used for dynamic analyses will be stored in the BLOB store.

3.3.3 Orchestrators and Enrichment Modules. We have opted for Python for implementing these. Many REs are familiar with this high-level language, and many REing tools provide a Python API. One can use any method supported by Neo4j to populate the KB with new data. The most performant method to import large amounts of data in bulk is Neo4j's built-in CSV importer.

3.3.4 KB Query Language. The standard way to interface with a Neo4j DB is through its own SQL-like Cypher query language. Cypher queries can be embedded in Python code as string literals, but those are rather hard to read and verbose, and need to be duplicated to be reused, which makes the use of Cypher in EMs and SOs cumbersome. Moreover, embedding Cypher queries in them would tie their code to a specific DB, which we want to avoid.

To alleviate these shortcomings, we designed our own query language inspired by the AST Matchers framework of the Clang compiler that is commonly used for tools operating on source code [43].

```
Instruction(
  writesTo(MemoryBuffer().bind("written_buf")),
  readsFrom(MemoryBuffer().bind("read_buf"))
).bind("ins")
```

Listing 1: Query combining node and traversal matchers.

```
Instruction(
  dependsOn(
    Instruction(
      P("opcode") == "add"
    ).bind("add_op")
  ),
  P("num_exec") == P("add_op", "num_exec")
).bind("ins")
```

Listing 2: Query illustrating both forms of the `P` function.

```
read_key_instruction_pattern = Instruction(
   readsFrom(
      MemoryBuffer(P("entropy") > 0.8).bind("key_buffer")
   )
).bind("read_key_ins")

Instruction(
   P("opcode") == "xor",
   dependsOn(read_key_instruction_pattern),
   dependsOn(
      Instruction(
         readsFrom(
            MemoryBuffer(
                P("entropy") < 0.2).bind("data_buffer")
         )
      ).bind("read_data_ins")
   )
)
```

Listing 3: A query illustrating increased reusability.

We hence borrow terminology from this framework where appropriate. Unlike Cypher queries, which are embedded string literals, our query language comes in the form of reusable Python code that is automatically converted to a Cypher query under the hood.

Our query language design revolves around reusable Python components called *matchers* that come in two types. *Node matchers* match one type of node. Examples are the `Instruction`, `BasicBlock`, and `MemoryBuffer` matchers. To create a node matcher, one simply calls the appropriately named function in Python: for example, `Instruction()` creates a node matcher for instructions. *Traversal matchers* match one type of relation in the KB, thus allowing traversal from one node to another. Each relation in the KB typically has two associated traversal matchers: one for each direction, such as a `readsFrom` traversal matcher (traversing from an `Instruction` to a `MemoryBuffer` that it reads from) and an `isReadFromBy` traversal matcher (traversing from a `MemoryBuffer` to an `Instruction` that reads from it). Matchers also support the `bind` operation, which gives a node or relation a name that can be used to refer to this node or relation later. For example, `Instruction().bind("ins")` matches instructions that can then be referred to by the name "`ins`". A node matcher can have zero or more traversal matchers as argument; a traversal matcher has one node matcher as argument. As such, node and traversal matchers can be combined arbitrarily to form complex patterns. For example, the query in Listing 1 looks for instructions that both read from and write to a buffer.

Node and traversal matchers can be narrowed with conditions on the properties of the nodes/relations they should match. This is done with built-in Python operators and the `P` function (short for property), which has two forms: `P("prop")` refers to the property `prop` of the node(s)/relation(s) matched by the encompassing matcher, and `P("name", "prop")` refers to the property `prop` of the node(s)/relation(s) bound to the name "`name`". The query shown in Listing 2 contains both forms to express that `add_op` must be an `add` instruction (Line 4) and that another instruction `ins` that depends on it must be executed the same amount of times (Line 7).

Besides being more human readable and Neo4j-independent, our language also makes it easier to reuse parts of a query, and to build a reusable query library. This is because information pertaining to the same artifact can be spread out over `MATCH` and `WHERE` clauses in Cypher, whereas it is more localized in our language. For example, the query in Listing 3 looks for artifacts in a crypto routine. It looks for XOR instructions that depend on two instructions that read from memory buffers: a buffer with high entropy containing the key, and a buffer with low entropy containing the plaintext data. Reusing the pattern for the instruction that reads from the key buffer is as simple as assigning that expression to a variable (Lines 1–5), and then using that variable in the query (Line 9).

3.3.5 KB Enrichment Modules. To demo our design and to promote its use, we implemented several EMs. Most of them are scripts that interact with the KB in an automated manner to enrich and populate it with knowledge. Some EMs are interactive, however.

Intel Software Development Emulator. Intel SDE is an executable instrumentation and emulation tool suite [22]. We use its instrumentation framework *Pin* for dynamic analyses implemented in *Pintools* [23]. We developed several EMs to collect dynamic information, such as data dependencies, call graphs, accessed memory buffers, etc. These EMs first record a program's execution in a *Pinball*. A recorded execution can then be replayed multiple times to run the actual Pintools. This offers the advantages of consistency (determinism) over multiple analysis runs, reproducibility, and improved tracing speed. The latter is important for tracing time-dependent app behavior, such as real-time or network apps that can time out. Note that the Pinballs and execution traces are considered BLOB data. The KB is only populated with actual knowledge in the form of data dependencies, call graphs, buffers with their statistical properties, executed instructions with their execution counts and which buffers they access, etc. For the example REing attack strategies, the developed Pin EMs collect all the information needed for reaching mileposts (a), (b), and (d) from Section 2, and most of the information needed for reaching milepost (c).

Dynamic Taint Analysis & Generic Deobfuscation. Yadegari et al. proposed a generic deobfuscator pipeline [50]. Forward and backward taint analysis are first performed on traces [49] to identify the executed instructions that are semantically relevant for producing the program outputs. The trace filtered for these instructions is then simplified based on whether or not the behavior of the instructions is so-called quasi-invariant. From the simplified trace, a CFG is then

reconstructed. These pipeline stages produce results of interest to REs. With our integration, they are stored in the KB, including the executed instructions, their taint status, and whether their operands are quasi-invariant. We can use this EM to identify those operands that originate from data input files for milepost (c).

Debuggers. With debuggers such as GDB [16] and LLDB [44], REs can execute a program to relevant points or activities, and inspect their state. They can do so interactively or with automated scripts. We created a template EM that can be instantiated to deploy the LLDB debugger with such scripts. This is facilitated by LLDB's built-in Python interpreter. Our template provides wrappers for the KB's APIs, such that the debugger scripts can at once control the debugger, and access and update the information in the KB. We implemented two such EMs, namely for reaching the example strategy mileposts (d) with breakpoints and (l) with watchpoints.

Disassemblers. Interactive disassemblers such as IDA [21], Binary Ninja [47], and Ghidra [28] are among the most used REing tools. They store their models of binaries in their own, custom DBs, which we treat as BLOBs. They can be accessed via custom, vendor-specific APIs, typically in Python. Importing the relevant information from the disassembler's DBs (call graphs, CFGs, cross-references between code and data, ...) into the KB is hence a simple enrichment process. We developed such an EM for mileposts (e)–(j).

Such disassemblers also offer plugin APIs for building custom analyses. We can hence build custom analysis EMs that combine information computed by the disassembler with information available in the KB. For example, we developed an EM to customize Binary Ninja's static Value Set Analysis (VSA) [2], on which the tool builds to identify realizable paths in CFGs. When the KB stores dynamic value sets previously obtained with a trace-based EM, this EM can invoke Binary Ninja, filter the computed static value sets (which are overapproximations) based on the dynamic sets, and then let Binary Ninja's own analyses refine the reconstructed CFGs based on the filtered value sets. While this improvement of Binary Ninja's operation is an unsound process, it is not uncommon for REs to simplify their mental model of a program based on properties observed dynamically. With this type of EM, such hybrid, static-dynamic knowledge gathering can be modeled easily. Appendix A provides additional illustration of this type of integration.

Pattern Matchers. While our query language facilitates searching in the KB, e.g., by matching patterns, we also designed EMs around two existing pattern matchers. The open-source Yara [48] helps malware researchers to identify and classify malware samples. Its users can define custom patterns (textual or binary) and find matches in files' contents. The open-source static analysis tool Grap [34] applies pattern matching on the CFGs of a binary. It uses the Capstone [11] disassembler to generate the CFGs that it searches for user-defined patterns. The matched patterns are then stored in the KB, where they can be linked to other data.

Scanmem. As an interactive EM, we implemented a wrapper script around the Linux scanmem tool [37]. It attaches scanmem to a running process; iteratively asks for values to search for in the process' address spaces, narrowing down the found locations in each step; and then stores the found locations in the KB. This EM can be used for milepost (k) of the example attack strategies.

Complexity Metrics. It is rather trivial to develop EMs that compute complexity metrics on the KB data, such as Halstead sizes and cyclomatic complexity code metrics [12], information flow metrics [19], etc. With a simple query, a subgraph can be extracted from the KB, which can then be imported into a Python script using the Python Object Graph Mapper of Neo4j, and then the script can compute the metric on the imported graph. Alternatively, one can program a small Java program with a specially crafted Cypher query to let Neo4j compute the complexity without having to serialize and then deserialize the data to pass it to Python.

Distance Metrics. Neo4j readily offers the necessary functionality to compute shortest distances between nodes in subgraphs of a graph. This allows to compute the distance between mileposts in the KB, and related information, such as the set of nodes that are closer to some milepost than another milepost. In Section 4, we will discuss how we propose to build on such metrics to model and estimate localization attack effort. Importantly, Neo4j offers the necessary functionality to first extract a subgraph, and then compute distances within that subgraph. This allows an EM or SO to compute distances assuming that certain forms of information in the KB would not be available to an attacker.

3.3.6 Inspection Interfaces. Beyond using Neo4j's web interface to inspect the KB, several options exist to export the information in user-friendly formats. First, Python scripts can be written (as separate EMs or in SOs) that compute and output aggregate data, such as statistics, selected pieces of knowledge (such as artifacts identified in a localization attack), or any other form of report. Furthermore, existing REing tools can be re-used to inspect information in the KB. For example, we developed a Binary Ninja plug-in to export data from the KB to Binary Ninja, and to augment the CFG visualization in Binary Ninja. When an instruction is selected in a displayed CFG, a dock lists the memory buffers accessed by that instruction when such information is available in the KB, together with their properties such as address, size, and number of accesses. It also shows the read/written values, with which VSA results can be refined as discussed above. In the CFG itself, the data dependencies of the selected instruction are highlighted as they occurred in collected program traces. Appendix A illustrates this further.

3.4 Demonstration on Example Strategies

For each of our example REing strategies, we implemented SOs that execute them by invoking and configuring the necessary EMs.

Cryptographic Keys. Two Python notebook SOs implement an attack template for two target apps. This template involves more than invoking REing tools sequentially. For milepost (a), the Pintool EMs are invoked for a number of differently sized inputs, and the results are stored in the KB. Based on that information, the SO then first selects the basic blocks that most likely form milepost (a), and then those that form milepost (b). Then it configures and invokes more Pintool EMs to collect additional tracing information. This can optionally include a whole-program trace as input to the taint analysis EM. The trace then is a BLOB, but the taint results are stored in the KB. It also includes fine-grained trace information on the operations that form milepost (b), which is also added to the KB. This requires customized Pintools, as collecting that information

on all code in the program would yield an unacceptable execution slowdown. Instead, the Pintools are configured to only trace the operations in milepost (b). Within those operations, and based on the additional information, mileposts (c) are then determined, after which a debugger EM is configured and launched to obtain the final milepost (d). We validated that this can fully automatically extract the AES keys used in 7-Zip [32] and in GnuPG [33] (both compiled not to use Intel's AES NI support, which would make key extraction trivial).

License Key Check. Our SO implements a version of the general attack strategy to target simple license check implementations. It is simplified in the sense that (i) it only relies on static information, all of which is imported into the KB by means of an IDA EM; (ii) it only targets simplistic implementations of license checks, namely where `int key = atoi(argv[i]);` extracts the key from the command-line arguments, and somewhere at random program points, there are checks of the form `if (key!=0x1234) {printf("invalid key\n"); exit(-1);}` in which all mileposts (f)–(j) are grouped together in the binary, i.e., in two adjacent basic blocks, rather than spread out.

This SO takes as input a list of terms one expects to find in license error messages. With the IDA EM, it collects the binary's strings, code references to those strings, and all CFGs. It then selects the string(s) that best fit the provided terms as milepost (e), and reports a list of basic blocks that refer to and feed them to functions such as `printf()`, as well as their preceding blocks that implement the key checks, assuming that these are the adjacent blocks holding mileposts (f)–(j). The actual extraction of the program slice that computes the check, either from the disassembled code, or from decompiled code, is left for the RE in this case. In a tool such as Ghidra, that offers built-in slice highlighting, this requires almost no effort in case the check is as simple as assumed here.

Game Resource Hack. For this strategy, a simple SO deploys two interactive EMs: a scanmem EM and a debugger EM. With the former, one obtains the memory location(s) of milepost (k) by performing actions in the running game intertwined with scanmem rounds. The SO then automatically configures LLDB to set watchpoints on those memory locations (while the game is still running), and to report all code fragments that access those locations.

4 Reverse Engineering Effort Meta-Model

The primary goal of our effort meta-model is to enable estimating the effort required to execute given REing strategies on given target apps by simulating their execution. For each strategy and target app, the meta-model will be instantiated into a concrete model, which will be nothing more or less than an SO and a KB.

We build on several assumptions for this endeavor. For starters, we consider REing strategies to involve four types of activities. These types of activities overlap mostly with the analyst's aims and means considered by Schrittwieser et al. [38], and with the activities in the taxonomy of Ceccato et al. [7].

(1) *Collection.* By creating, building, customizing, configuring, and executing software analysis tools on their targets, REs obtain the four kinds of information mentioned in Section 3.1.
(2) *Localization.* With tools, but not necessarily automatically, attackers localize the artifacts (i.e., code or data) of interest.
(3) *Comprehension.* Attackers try to comprehend that information, and learn certain concrete or abstract properties of artifacts, such as invariants or summaries of semantics, as well as relations, and which lower-level constructs map to which higher-level concepts, by studying them manually.
(4) *Strategy building.* Based on the already obtained information, attackers dynamically develop their strategy, i.e., decide on the next activities to perform. These decisions include performing manual tasks, invoking more data collection tools, performing manual comprehension tasks, as well as formulating hypotheses, testing them, and refuting them.

The first three of these activities can be modeled as enrichments of the type of KB proposed in the previous section. This is obvious for collection activities and for comprehension activities, the latter merely being the manual counterpart of automated software analysis. Localization activities can be modeled as the ordering of information in the KB. Their outcome is a list of candidate nodes ordered by some priority function. These nodes can be nodes that already existed in the KB, such as functions or basic blocks, but it can also be newly created nodes that map (via the *mappings* discussed in Section 3.1) onto arbitrary types of subgraphs of the KB graph. For example, each item in the list can be a newly created "code region" node with a corresponding set of basic blocks. The outcome of a localization activity can hence also be stored in the KB, in the form of nodes, mappings, and properties, with the latter storing the priority assigned to each node by the priority function.

This framing of localization might seem counter-intuitive. For example, when a RE browses a function's CFG in a disassembler's GUI to find some part of interest, studying the displayed basic blocks is typically considered part of the localization effort. We consider the studying of the basic blocks and the CFG comprehension tasks, however, and limit localization to the process of prioritizing where to look next. This allows for a cleaner partitioning of the different types of activities to be modeled, and how to model them.

The strategy building activity can be modeled as a function that takes as input the state of the KB and that outputs the next activity to be executed, which will either be one of the first three activities or the *halt* activity that halts the REing process. This activity will be triggered either when an amount of effort has been invested at which the RE is assumed to give up, or when all mileposts have been discovered according to the information in the KB, i.e., when the mappings in which the RE is interested, have all been found.

A whole REing process can then be modeled as an infinite loop:

```
while (true)
   activity, subgraph = strategy(kb, effort)
   k, e = execute(activity, subgraph, kb)
   kb = k; effort += e
```

In this meta-model pseudo code, `strategy` is the strategy building function. It returns the next activity to execute, such as some data flow analysis, as well as the subgraph of the KB on which that step will be executed, such as the function and its CFG. The function `execute` simulates the next activity on the subgraph and returns an updated KB and the effort required to execute the step.

Critically, both `strategy` and `execute` are probabilistic in nature, as will be discussed in detail below. To estimate the expected effort for a REing strategy, two options are then available: analytical

(statistical) analysis, and simulation. Here, we focus on the latter. The main idea is to simulate the REing process many times, using random number generators where probabilistic processes take place, thus obtaining a distribution of the required effort. Our proposed simulation approach is to implement the above script in an SO. The main challenge then is of course the scripting of the various possible activities and how those simulate real REing activity executions.

Before we discuss that in detail, we need to point out that the execution of activities can add additional information to the KB beyond information about the target app. More specifically, the execution of an activity can record information in the KB about the REing process itself. Three examples of such useful information, on which REs in practice rely to decide on their next activities, are (i) the effort invested while executing an activity on a subgraph, (ii) confidence estimations of obtained information, (iii) likelihood of relevance. We come back to the use of these later in this section.

Furthermore, we assume that before starting a strategy simulation, the KB will have been populated with all relevant ground-truth data. This builds on the assumption that the modeler has access to that data, which in practice limits the use of our approach to developers wanting to model REing attacks on their own software or on software that they were able to analyze sufficiently themselves, e.g., through source code access and assuming that the used build tools produce sufficient logs, symbol information, debug information, and other metadata. This is not trivial, but feasible [31].

Importantly, that ground-truth data initially added to the KB will be labeled as "hidden". In all models of activities that get simulated, hidden information can be used to simulate them correctly, but it cannot be considered available to the RE. For example, in an activity visiting and inspecting a partially ordered set of artifacts to find the relevant one, the model can consider the ground-truth position of that one artifact to compute the expected number of visits, but it should not assume that the RE uses this knowledge to optimize the order of their visits.

4.1 Collection Activities

Tool Execution. Several options exist to model data collection with automated tools. If the modeler has access to all tools the target RE can potentially use, their execution can be simulated by invoking corresponding EMs that add the tools' relevant outputs to the KB and report their execution time. If the RE can choose among multiple comparable tools for a task (e.g., multiple disassemblers can extract a call graph or all string references) and the modeler has access to all of them, the EM can make a random selection. Alternatively, the modeler can opt to use only one of them if it can be considered representative for the comparable tools. This loosens the requirement that a modeler needs access to all possible tools and tool versions that the target RE might be using.

When the modeler lacks access to some tool and comparable alternatives, another option is to rely on literature about the scalability of the tool's analyses and on ground-truth data about the target binary. That data serves two purposes. First, rather than having the EM populate the KB with information extracted with the modeled tool, the EM can uncover the corresponding ground-truth information that the tool was aiming to extract, by removing the "hidden" label from that information in the KB. This overestimates the capabilities of the RE and their tools, as this corresponds to a worst-case scenario assumption that the RE's tool can extract the ground truth accurately. In practice, this is not the case due to unsound analyses and heuristics being used that can result in false positive and false negative results (e.g., [31]). Overestimating the RE's capabilities will result in underestimating the required effort, so uncovering ground-truth data as an alternative will still allow the modeler to obtain a lower bound on the expected effort. Secondly, the ground-truth information can be used to obtain the properties of the binary to which the analyses' execution times are sensitive, and thus to estimate the analysis tools' execution times.

Tool Preparation. Custom tool development effort can be estimated based on complexity aspects of their analysis, such as locality (local, global, interprocedural, interthread, interprocess) and sensitivity (flow, path, context, call-site, ...). It can also be neglected, which comes down to the worst-case scenario assumption that expert REs readily have the tools available. Similarly, the configuration effort might be neglected, under the worst-case assumptions that REs are highly productive experts with automated aids (e.g., scripts) for the configurations. For example, not to underestimate one's adversaries, it might be wise to assume that they have access to the same SOs and EMs the modeler is using. Note that the stated worst-case assumptions about the targeted RE's capabilities and expertise do not prevent the modeling of less capable adversaries, such as script kiddies or amateur hackers. If certain tools or activities are too complex for the supposed adversary, as might be the case for custom tool development, the modeler can simply model strategies that exclude those tools and activities entirely.

4.2 Comprehension Activities

Obtained Information. We have already stated that we assume comprehension activities to produce new information. Users of interactive disassemblers do so quite literally, e.g., when they rename meaningless symbols such as `DAT_0x1234` and `FUN_0xdcba` to useful ones such as `key_buffer` and `ht_insert`. But also recognizing properties, relations, or mappings can be outcomes of human comprehension activities that correspond to forms of information added to the KB. The successful execution of a comprehension activity can hence be simulated by injecting the obtained knowledge into the KB. If this knowledge corresponds to hidden ground-truth data already in the KB, it suffices to remove the "hidden" from it.

We note that one can model that a comprehension activity executed on one subgraph probabilistically results in knowledge on related fragments. For example, we have observed that REs navigate to one code fragment in a disassembler GUI to study it, and then observe that the actual fragment they are after is right above it. Our models are flexible enough to model such serendipitous discoveries.

Required Effort. To model and simulate comprehension effort, we propose to model the required time as a Gaussian distribution, called the success distribution, of which the parameters μ and σ^2 are a function of the complexity of the subgraph being studied. Remember that this subgraph is determined in the strategy building activities, which will be discussed below in Section 4.4. We assume that the subgraphs' complexity (which can be a multi-dimensional value, e.g., if the comprehension requires studying different aspects

such as CFGs as well as data dependencies) can be computed by means of complexity metrics (as is commonly done in the field of software engineering research), and then reduced to μ and σ^2. Determining which metrics are best suited, and how to reduce them, is out of scope of this paper, but inspiration can be found in many places [12, 19, 26, 27]. More complex statistical distributions can also be considered, but studying those is future work.

Partial and Iterative Comprehension. In practice, REs do not always complete comprehension activities successfully. After some time, they can give up to try an alternative strategy, or change priorities and shift their attention to other candidates, and potentially revisit fragments again later. In short, REs take into account how the process is going and has been progressing earlier on. It is for enabling such considerations in the models that we previously mentioned that the simulation of activities can also produce, and record in the KB, outcomes such as the invested effort, confidence estimations, and likelihood of relevance. These can be attached as properties to the subgraphs on which an activity is executed.

The model of a comprehension task can then, e.g., be parametrized by a stubbornness parameter that models how long the RE will persist before giving up. This parameter might be a simple constant, or a Gaussian distribution, that may be conditional on the information in the KB. Every time the activity is executed, a sample taken from that stubbornness distribution determines probabilistically how long the RE will invest maximally in this execution. Then the success distribution is sampled. Considering the time already spent on the subgraph in previous executions of the same or similar activities, the two samples determine whether the current execution succeeds or fails, i.e., whether or not the aimed for information is added or unhidden in the KB, and the invested (total) effort is recorded in the KB as well. That recorded effort can then later be used in the priority functions used in localization activity models, e.g., to allow correct modeling of the fact that other, not yet visited candidates will be visited before the same candidate is revisited.

Confidences or likelihoods produced in comprehension activities can then also help later localization activities prioritize subgraphs to be (re)visited. For example, in experiments with the license key check, some REs visited all code fragments that access the stored key. Assuming that the app might contain decoys, they first briefly studied those fragments to determine which ones are likely true checks and which ones are more likely decoys, after which they revisited the former to perform more complex manual analysis. The first visit can be modeled with an exploration comprehension activity, in which the RE invests a fixed amount of time to record a likelihood in the KB, while the second is a deep comprehension activity that aims for ground-truth uncovering. How to compute such likelihoods, and how to make other distributions dependent on them, is out of the scope of this paper. It is highly activity-specific. Our main claim here is that if the modeler is familiar with all the constructs and relations that a RE will consider for decision making, and how they will do so, it is possible to model those processes, and to formalize them in executable EMs and/or SO scripts.

Note that determining likelihoods, confidences, and other constructs that can steer the strategy execution should not be limited to comprehension tasks. While we discussed this aspect here, collection tasks can obviously be used for that purpose as well.

4.3 Localization Activities

As explained in the introduction of Section 4, we consider localization activities to be operations that assign priorities to subgraphs in the KB, potentially creating those subgraphs on the fly. Those priorities can then be taken into account by the strategy building activities that will be discussed in the next section.

Our meta-model does not prescribe a specific type of priority function for modeling such activities. In practice, a wide variety of heuristics are used, and REs navigate through graph representations of programs in various ways. The example strategies from Section 2 to identify likely mileposts based on all kinds of properties and relations provide only a small sample of all possible fingerprints and relations that REs rely on in practice. In general, if one can formalize the heuristics and express them in a script, they can be incorporated in our approach. There are some interesting cases and aspects, however, which we wish to highlight.

First, it is up to the modeler to decide on the granularity of the localization activity outcomes, i.e., which subgraphs are prioritized. To model a RE browsing through code trying to identify the next milepost starting from an already found one, the localization activities might prioritize functions to visit, or basic blocks, or more general code regions, etc. The modeler is free to choose this, as long as the comprehension activities that then model all visits of the browsing process handle corresponding types of subgraphs.

Secondly, it is possible for a localization activity to prioritize subgraphs hierarchically, and to take those hierarchical priorities into account inside comprehension activities, e.g., to model manual browsing through graphs. For example, a localization activity can prioritize and construct a single code region, and order the basic blocks in that code region to model the likely order in which the RE will browse them. Or it can prioritize one part of a program's call graph, and order the functions in it. This can be used, e.g., for REing strategies in which the RE has identified a milepost function, and goes searching for a related function in the call graph. The RE can then browse blocks or functions randomly, but also with some strategy. For example, Mantovani et al. studied depth-first, breadth-first, and other code browsing strategies [26]. To implement priority functions that correspond to such strategies, the distance metrics and Neo4j's capabilities to compute shortest distances in Section 3.3.5 are particularly helpful. For example, it offers a trivial way to compute how many nodes or functions will likely be visited before the relevant one will be visited. Which the relevant one is, is of course assumed to be available in the ground-truth information.

A modeler can choose whether to use hierarchical localization and comprehension models. Determining which approaches work best for which types of activities is future work.

Furthermore, we should point out that it is by no means necessary for localization activities and the used priority functions to produce and use total orderings. Any type of function will do, even simple binary classifiers that partition the KB in relevant and irrelevant parts are fine. The next section discusses how to handle cases where multiple subgraphs are assigned the same priority.

Finally, with respect to the effort that localization activities require, we conjecture that their execution times can simply be measured or estimated, similarly to how the execution times of analysis tools invoked for data collection can be measured or estimated.

4.4 Strategy Building - Decision Making

The model for strategy building is one big function that encapsulates all the domain-specific knowledge that the RE will rely on during the execution of the REing strategy. It scripts the whole strategy from one milepost to another. The fact that mileposts have been found is made visible in the KB because the relevant ground-truth information will no longer be hidden. When all mileposts have been uncovered, the halt activity gets invoked.

The strategy building function is probabilistic when the priority functions of localization activies are not guaranteed to return unique priorities for each subgraph as input to the next activity. When multiple subgraphs can be chosen with an equal top priority, a random selection will be made in each simulation of this activity.

We assume the strategy building itself requires no effort, under the assumption that the RE is experienced enough to decide on the spot as soon as all the relevant information is available. In other words, we assume that all time-consuming aspects of decision making are modeled in the other types of activities that gather the necessary information, including the prioritizations.

4.5 Demonstration on Example Strategies

Cryptographic Keys. The SOs introduced in Section 3.4 automatically identify AES keys in our sample programs, so their required effort can simply be measured. This revealed that almost all time is spent collecting traces. We then performed an ablation study in which we simulate attacks that skip the collection of certain forms of trace information to check if saving on tracing time could be worthwhile. Some prioritization heuristics then become unreliable of course, which can prolong later attack steps (including the collection of detailed tracing info on selected code fragments). To avoid this, an attacker might manually try to filter out irrelevant intermediate mileposts. We modeled this by means of (i) a localization activity for ordering candidates to be visited; (ii) a comprehension activity for visiting them. We assume that each comprehension visit correctly labels a candidate as relevant or irrelevant (as indicated by ground-truth data), and that the effort required to do so depends linearly on their Halstead difficulty [17].

The summary outcome is that, in case the attacker is assumed to have enough resources to perform all tracing required for some milepost concurrently, and in case manually filtering a block as irrelevant requires at least a couple of seconds, which we deem a realistic assumption even for assembly experts, using all trace information except detailed taint tracking is the most efficient strategy.

License Key Checks. For modeling attacks on these, we consider attackers that expect the program to contain decoys of the form `if (OP && subkey1!=0x654) {printf("invalid key");exit(-1);}` in which OP is an opaque predicate [10], i.e., a value that is computed dynamically but always evaluates to false, such that the check always "succeeds". We extended the key check SO introduced in Section 3.4 to model an attacker that visits each potential key check to assess whether it is preceded by an OP that makes that key check a decoy, assuming that the attacker relies on their experience to recognize OPs. The effort required to do so is (in this simple model) again assumed to depend linearly on the Halstead complexity of the basic block in which the predicate is computed.

With this model, we can obtain distributions for the expected effort of different strategies by simulating those on a target app that contains various true subkey checks and various decoy subkey checks. These strategies can differ, e.g., in whether or not the attacker continues checking until all of them have been visited, or stops when the first true check has been identified.

This model and the license check implementations that it targets, are of course of a contrived simplicity. In ongoing work, we are modeling more complex strategies and more complex models (e.g., using more relevant complexity metrics) targeting more complex license checks in which, e.g., the OPs can be spread throughout the code, and the references to the strings from the code are also moved around to create a larger distance between them and the key checks, thus making the attack more difficult. In future work, we plan to validate those models by comparing them with how our Master Computer Science students attack those implementations in labs of a course on SP and hacking.

Game Resource Hack. The scanmem and debugger EMs introduced in Section 3.4 are both interactive and stochastic. They require playing the game while performing analyses, and how cheaters would play the game is obviously not deterministic. For example, the points in time at which they would perform scans can vary, and hence the outcome of that attack step can vary. In simple strategies, on games in which the resource data is not obfuscated, that outcome will likely not vary: the cheater finishes the scanmem process when exactly one memory location has been identified, and this will be a true positive for milepost (k). If the resource data would be obfuscated, however, cheaters will use scanmem in a more sophisticated manner, involving many more scans and more sophisticated heuristics to prune irrelevant memory locations, which might end with a set of potentially relevant locations. Which false positives this set includes will depend on which bytes in memory accidentally happen to have the same value at the point of each scan. The input to the debugger phase in the form of a set of memory locations hence has to be modeled stochastically.

Obviously the modeler cannot be asked to play the game over and over again for the repeated simulation of these processes. Instead, we propose to have the modeler execute the game once and to collect sufficient information during that one run to simulate multiple, randomized runs. Just like the strategy, this one run would consist of two phases of gameplay. In the first phase of that one game run, we propose that the modeler takes a much larger set of memory dumps than would normally be necessary for a cheater. This then allows repeated randomized simulations of the scanmem attack step by drawing random dumps from that set and performing the scans on those dumps. The modeler then pauses their game play, to first perform those randomized simulations of the scanmem step. The result will be a set of sets of memory locations, i.e., one set of locations for each simulation run. The union of these sets is then injected into a debugger EM script for the second phase, in which the modeler then continues playing the game while the script collects candidates for milepost (l), i.e., code fragments that access the potential resource data. The script does so by rotating through the whole set of memory locations and setting watchpoints to them. By rotating through them, information can be collected on more than four watchpoints, which is the maximal number of hardware

watchpoints supported on modern x86/IA64 architectures. This way, in one run of the game, all the necessary information is collected to simulate the debugger attack step repeatedly, i.e., once for every scanmem simulation performed in the first phase.

4.6 Discussion

We consider our meta-model and simulation approach flexible and expressive enough for modeling real-world REing strategies, but complex to use. We consider this unavoidable, because REing processes are complex and cover a wide range of constructs, models, methods, and instantiations thereof. With this paper, we launch a repository of SOs and EMs that can instantiate the meta-models for our example REing strategies, for reuse in the SP and REing research communities. It is our hope that the availability of this repository will eventually lead to better evaluations in scientific papers on these topics. If the repository grows in the future, incorporating more and more models of REing activities and strategies, it should become easier and easier over time for researchers to instantiate and use the relevant models for their evaluations.

5 Reverse Engineering Strategy Capturing Tools

To use the effort meta-model from the previous section, the modeler needs to know the REing attack strategy to model. Knowledge of commonly used strategies can be obtained in various ways, including from the scientific literature and reports from practitioners [7]. Tools have also been proposed to automatically collect which code fragments are being visited at which times during the execution of a REing assignment [26, 51], thus gaining insights into the order in which REs visit code fragments. Such tools can help modelers in gathering knowledge about attack strategies, but clearly they do not capture all relevant information. Specifically, they do not track how REs navigate from one fragment to another, i.e., which relations they exploit that could be modeled in a KB and that should be considered in an attack effort model. Furthermore, they do not track which comprehension results the REs obtain. To fill these gaps, this section proposes novel methods and tools to extract how REs make use of code search navigation tools and relations between artifacts, as well as how they gather comprehension results from tool-neutral data collected with a generic data collection tool.

Our tools build on data collection tools such as RevEngE that run in the background on the system on which the participating RE executes his strategy [41, 42, 51]. The collected, time-stamped data includes screenshots of sufficient quality to obtain accurate optical character recognition (OCR) results, mouse clicks with coordinates, key logs, active process information, and active window data.

As is the case in the most closely related research [26, 51], we assume that the researcher wanting to extract strategic activities from the collected data has access to ground-truth information on the operation of the tools used by the participating RE. In our research, we so far focused on extracting strategic activities involving interactive disassemblers such as IDA Pro, Binary Ninja, Ghidra, etc. After their initial analysis of a targeted app, these tools choose names for artifacts found in the app, such as `DAT_0x1324` and `FUN_0xffca` for global data and functions found at the indicated addresses. Our assumption is that the researcher can extract these symbols, as well as the whole internal (graph-like) representation of the program from the disassembler's custom DB, as discussed in Section 3.3.5. The researcher (and the extraction tools we discuss below) hence know which symbols to expect in screenshots and which relations the tools know about that can be exploited by their users to navigate or search in the binary.

5.1 Extracting Activities with Keyboard Inputs

Several localization and comprehension activities map directly onto the use of features of the disassemblers that involve text input from the RE. Appendix B presents some screenshots thereof. A *Search* feature supports localization activities by allowing engineers to find specific symbols or strings within the code and helping them quickly localize relevant mileposts. The use of the *Rename* or *Label* features can be seen as a method with which the RE persistently stores information obtained through comprehension or localization activities in the disassemblers DB, and hence marks the end of such activities. A *View References* feature facilitates localization by providing the REs with a list of all references to a symbol, thus allowing them to trace the flow of data and/or control.

For extracting these types of activities from the collected data, our tools combine 4 steps. First, using the OCR'ed screenshot data, our tools determine which (sub)window is active, and with it which navigation or search functionality of the disassembler is being used. Secondly, the tool determines the time frame in which that window is active, and hence receiving keyboard input. Thirdly, the keystroke data from within that time frame is retrieved. Fourthly, if relevant, the tool determines on which artifact or symbol the window was opened. For example, for a renaming activity, the tool determines which symbol is being renamed. Two options are available for this. The first option is to extract the symbol from the active window using OCR. We found this option to be unreliable, because OCR often does not perform well on such symbols due to the way they are being displayed with, e.g., little contrast. The alternative option is to determine which symbol was being clicked to open the window. The implementation of that option is discussed in the next section.

These steps suffice to report time-stamped activities such as "searched for string `invalid`" or "renamed `bVar8` to `keycheck`."

5.2 Extracting Mouse Click Activities

Navigation through displayed artifacts and opening windows to perform activities is often done with mouse clicks in interactive disassemblers. Sometimes single or double clicking suffices to navigate directly to another artifact, sometimes it opens a pop-up menu from which an item is selected with another click.

To identify which symbol or menu item is being clicked, our tools use the mouse info, including the coordinates of the click, as well as the OCR'ed screenshots. More specifically, OCR tools such as Tesseract can produce hOCR files, which are standardized XHTML files describing, among others, the bounding boxes in the images of the retrieved text fragments. With the click coordinates, the relevant bounding box is determined, and hence the text displayed in it. If that text contains more than one clickable symbol or menu item, the relative position of the click within the bounding box is used to select the most likely clicked symbol or item, respectively.

Existing work has already shown that tools can also determine which functions or basic blocks are shown on screen in tools such

as IDA Pro and Ghidra [51]. We build on such tools to determine the context in which an identified symbol is being clicked. This forms the source of the navigation activity. The target of the activity is the artifact denoted by the symbol itself.

Several options exist to determine which type of relationship between source and target artifacts is being used for direct navigation. Disassemblers typically encode that information in the text colors, and/or in suffixes or prefixes of the symbols. For example, cross-references are drawn in green in Ghidra's Listing window, and followed by `(W)`, `(R)`, or `(*)` to denote read, write, or pointer references. These "clues" can be extracted from the screenshots, and they provide sufficient information: if they would not do so, their meaning would not be obvious to the RE using the tool either.

Together, the contexts before, between, and after the clicking, the clicked symbols and menu items, and the extra clues suffice to detect and report time-stamped activities such as "navigated from data `DAT_00288bb4` to function `FUN_001bed20` using (R) cross-reference."

5.3 Demonstration on License Checks

The following sample output demonstrates the information capture by one of our tools:

```
14:37:48, Symbol: FUN_0010ed40, double click
14:37:48, Entered Function: FUN_0010ed40
14:38:09 – 14:37:57: Feature: Rename Function, Word: main
14:39:52, Symbol: DAT_00288bb, single click
14:39:54 – 14:40:02: Feature: Edit Label, Word: keyplus0x1000
14:40:10, Symbol: keyplus0x1000, single click
14:40:13, Symbol: Find References to keyplusOxl000, single click
14:40:15 – 14:40:18: Feature: References to
14:40:26 Entered Function: FUN_001A3A20
14:40:26, Symbol: bVar8, single click
14:40:28 – 14:40:29: Rename local variable, Word: license key
```

Each entry is a timestamped activity, such as mouse clicks, keystrokes, and context information. In this snippet, several steps illustrate the key localization technique used by the RE. At `14:37:48`, a double click on `FUN_0010ed40` indicates the engineer entered this function, followed by renaming it to `main`. The RE interacts with the symbol `DAT_00288bb` at `14:39:52`, renaming it to `key_plus0x1000`, indicating a process of identifying and labeling a critical symbol for their strategy. At `14:40:13`, the RE uses a "Find References" feature to locate occurrences of `keyplus_0x1000`. This step is crucial for understanding how and where this key is used within the code. Finally, the engineer enters `FUN_001A3A20` at `14:40:26` and renames the local variable `bVar8` to `license key`, when the license key was finally found. These snippets of extracted information can easily be assembled, using pattern matching, into activity reports like those reported at the end of the Sections 5.1 and 5.2. This then shows the strategy taken by the RE to localize a milepost.

6 Related Work

Several REing "frameworks" and tools have been proposed in the past, all of which aim to facilitate R&D into REing and binary analysis techniques, including disassembly, decompilation, and static and dynamic analysis techniques, by means of a flexible, effective, and extensible design that often involves lifting binary code to one or more custom IRs. Examples are commercial offerings such as Binary Ninja [47], IDA Pro [21] and Hex-Rays decompiler [20], as well as open-source ones such as Ghidra [28], angr [39], Capstone [11], Jakstab [24], Radare2 [35], Rizin [36], Amoco [45], Miasm [15], BINSEC [3], BARF [18], and BAP [4]. Among these, angr has been put forward as a unifying framework for REing. In practice, however, this has not resulted in unification or standardisation of REing research, as observed by De Sutter et al. [40]. In practice and in scientific literature, many different tools and frameworks keep being used. Combined with the fact that they are most often not compatible, each having their own focus, strengths, and weaknesses, this makes it hard to study (human) REing, to measure the effect of SPs on REing tasks, to compare the added value of scientific contributions over existing work, etc.

Our framework, models, and tool support do not aim to replace the listed tools. They instead aim for creating an environment/toolbox/methodology in which they can easily be combined—in the way practitioning REs combine a myriad of tools—and used for estimating the effort required to execute given REing strategies on given apps, and to compare the required efforts for different strategies, or on differently protected app versions.

Earlier attempts at creating such toolboxes have had a limited scope, such as Argon [1] that only combines one SP tool, Tigress [9], with two analysis tools, KLEE [5] and angr [39].

Our tools to capture strategic REing activities build on the existing RevEngE [41, 42] and reAnalyst [51] tools to collect data from the systems on which REs execute their strategies, to visualize that data, and to annotate it manually or through automated extraction of information from the collected data streams. The tools proposed here extend the types of activities and annotations that can be extracted automatically. Compared to ReMind [26], another tool to study how REs approach their tasks, our tools offer the advantage of being tool-neutral. They work for IDA Pro, Ghidra, etc. instead of being tied to a self-developed disassembler mockup like ReMind.

7 Conclusions and Future Work

We presented a meta-model to structure and store all information that REs collect about their target. We also presented a meta-model of which instantiated models can be simulated to estimate the effort that REing strategies require on given targets. Finally, we presented methods to extract strategic activities from data streams collected during human REing experiments. In support of these models, we presented and open-sourced a range of tools support. We demonstrated their use on three example REing attacks.

All of our models and tools are available at https://github.com/csl-ugent/TREX and https://github.com/csl-ugent/reAnalyst.

Our aim is that these repositories of models and tools will grow in the future, thus making it easier and easier for researchers in MATE SP to evaluate how SPs impact attacks in an acceptable, standardized, reproducible manner. We certainly plan to keep adding models and tools in support of our own future evaluations of SPs and attacks thereon.

Acknowledgments

This work was supported in part by National Science Foundation grant 2040206, by the Cybersecurity Research Program Flanders, and by Research Foundation - Flanders (FWO) grants 3G0E2318 and 11I1123N. Waldo Verstraete helped us with the game use case.

References

[1] Deepak Adhikari, J. Todd McDonald, Todd R. Andel, and Joseph D. Richardson. 2022. Argon: A Toolbase for Evaluating Software Protection Techniques Against Symbolic Execution Attacks. In *Proc. SoutheastCon 2022* (Mobile, AL, USA). IEEE, 743–750. https://doi.org/10.1109/SoutheastCon48659.2022.9764028

[2] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. 2005. CodeSurfer/x86—A Platform for Analyzing x86 Executables. In *Compiler Construction*. 250–254.

[3] binsec 2024. https://binsec.github.io/.

[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*. 463–469.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Symposium on Operating Systems Design and ImplementationOSDI*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[6] Nick Cano. 2016. *Game hacking: developing autonomous bots for online games*. No Starch Press.

[7] M. Ceccato et al. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering* 24, 1 (2019), 240–286. https://doi.org/10.1007/s10664-018-9625-6

[8] Cheat Engine 2024. Cheat Engine. https://www.cheatengine.org/.

[9] Christian Collberg. 2024. The Tigress C Obfuscator. https://tigress.wtf/.

[10] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/268946.268962

[11] COSEINC. 2024. capstone: The Ultimate Disassembly Framework. https://www.capstone-engine.org.

[12] B. Curtis, S.B. Sheppard, P. Milliman, M.A. Borst, and T. Love. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering* SE-5, 2 (1979), 96–104. https://doi.org/10.1109/TSE.1979.234165

[13] Bjorn De Sutter, Christian Collberg, Mila Dalla Preda, and Brecht Wyseur. 2019. Software Protection Decision Support and Evaluation Methodologies (Dagstuhl Seminar 19331). *Dagstuhl Reports* 9, 8 (2019), 1–25. https://doi.org/10.4230/DagRep.9.8.1

[14] Biniam Fisseha Demissie, Mariano Ceccato, and Roberto Tiella. 2015. Assessment of data obfuscation with residue number coding. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 38–44.

[15] Fabrice Desclaux. 2012. Miasm: Framework de reverse engineering. *Actes du SSTIC. SSTIC* (2012). https://github.com/cea-sec/miasm.

[16] Free Software Foundation. 2024. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb.

[17] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[18] Christian Heitman and Iván Arce. 2014. BARF: a multiplatform open source binary analysis and reverse engineering framework. In *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*. https://github.com/programastic/barf-project.

[19] Sallie Henry and Dennis Kafura. 1981. Software structure metrics based on information flow. *IEEE transactions on Software Engineering* 5 (1981), 510–518.

[20] Hex-Rays. 2024. Hex-Rays Decompiler. https://hex-rays.com/decompiler/.

[21] Hex-Rays. 2024. IDA Pro: A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro.

[22] Intel Corporation. 2024. Intel Software Development Emulator (Intel SDE). https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html.

[23] Intel Corporation. 2024. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[24] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Computer Aided Verification*. 423–427.

[25] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018-10-15). 412–425. https://doi.org/10.1145/3243734.3243783

[26] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. RE-Mind: a First Look Inside the Mind of a Reverse Engineer. In *31st USENIX Security Symposium*. 2727–2745. https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani

[27] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

[28] National Security Agency. 2024. Ghidra. https://ghidra-sre.org.

[29] Neo4j, Inc. 2024. Neo4j Graph Data Platform. https://neo4j.com.

[30] Neo4j, Inc. 2024. What is a Graph Database? https://neo4j.com/developer/graph-database.

[31] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. 2022. Ground truth for binary disassembly is not easy. In *31st USENIX Security Symposium (USENIX Security 22)*. 2479–2495.

[32] Igor Pavlov. 2024. 7-Zip. https://www.7-zip.org.

[33] The GnuPG Project. 2024. The GNU Privacy Guard. https://gnupg.org.

[34] QuoSec GmbH. 2024. grap: Define and match graph patterns within binaries. https://github.com/QuoSecGmbH/grap.

[35] Radare Developers. 2024. radare: Libre and Portable Reverse Engineering Framework. https://rada.re/n/radare2.html.

[36] rizin 2024. rizin - Free and Open Source Reverse Engineering Framework. https://rizin.re/.

[37] scanmem 2024. Scanmem. https://github.com/scanmem/scanmem.

[38] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (apr 2016), 37 pages. https://doi.org/10.1145/2886012

[39] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[40] Bjorn De Sutter, Sebastian Schrittwieser, Bart Coppens, and Patrick Kochberger. 2024. Evaluation Methodologies in Software Protection Research. arXiv:2307.07300 [cs.CR]

[41] C. Taylor. 2022. *Remotely Observing Reverse Engineers to Evaluate Software Protection*. Ph. D. Dissertation. The University of Arizona.

[42] C. Taylor and C. Collberg. 2019. Getting revenge: A system for analyzing reverse engineering behavior. In *Proc. Malware Conference*.

[43] The Clang Team. 2024. Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html.

[44] The LLDB Team. 2024. The LLDB Debugger. https://lldb.llvm.org.

[45] Axel Tillequin. 2024. https://github.com/bdcht/amoco.

[46] Jens Van den Broeck, Bart Coppens, and Bjorn De Sutter. 2021. Obfuscated integration of software protections. *International Journal of Information Security* 20, 73–101 (02 2021). https://doi.org/10.1007/s10207-020-00494-8

[47] Vector 35. 2024. Binary Ninja. https://binary.ninja/.

[48] VirusTotal. 2024. YARA. https://github.com/VirusTotal/yara.

[49] Babak Yadegari and Saumya Debray. 2014. Bit-Level Taint Analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (Victoria, BC, Canada, 2014-09). IEEE, 255–264. https://doi.org/10.1109/SCAM.2014.43

[50] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy*. 674–691. https://doi.org/10.1109/SP.2015.47

[51] Tab Zhang, Claire Taylor, Bart Coppens, Waleed Mebane, Christian Collberg, and Bjorn De Sutter. 2024. reAnalyst: Scalable Analysis of Reverse Engineering Activities. arXiv:2406.04427

A Inspecting KB data in Binary Ninja

Sections 3.3.5 and 3.3.6 discussed briefly how information stored in the KB can be used and inspected in Binary Ninja. Here we elaborate on and illustrate that.

Figure 2 shows a screenshot of Binary Ninja with our integration plugin installed. The dock ① on the top right lists the memory buffers accessed by the currently selected instruction, and their properties such as address, size, and number of reads/writes. The dock ② at the bottom right lists the read/written values of the currently selected instruction. In the assembly CFG ③ on the left, the dependencies of the currently selected instruction are represented by a red highlight, and the values read by the instructions are shown as comments. It is also possible to show the DDG of the entire function (not shown in the figure).

In the example CFG, the jump `je .skip_move` jumps to one of two basic blocks, depending on whether the read value is zero or non-zero. Since Binary Ninja initially had no information on this value, both directions are shown as possible paths in the assembler CFG as well as in the high-level intermediate language (HLIL) representation of the code (④) on the left of Figure 3. This is the HLIL

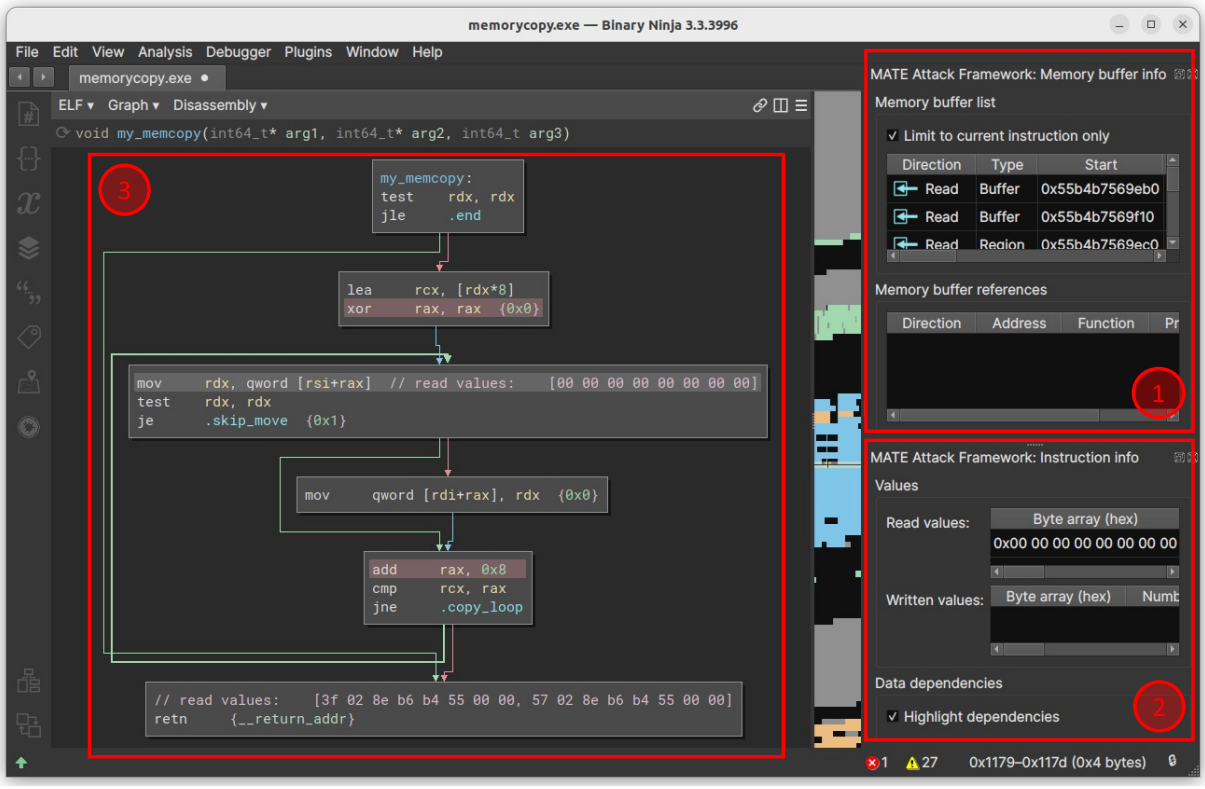


Figure 2: Screenshot of Binary Ninja illustrating how dynamic analysis results from the KB that were collected with Pintools can be integrated into this static REing tool and presented to the user.

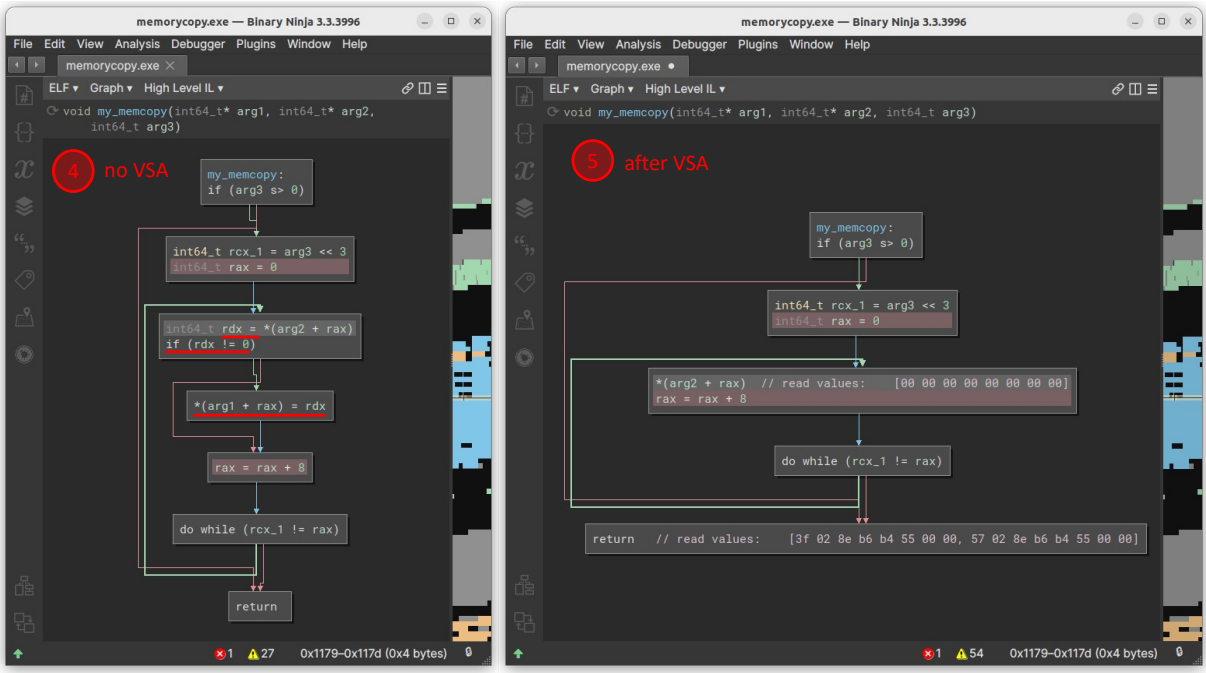
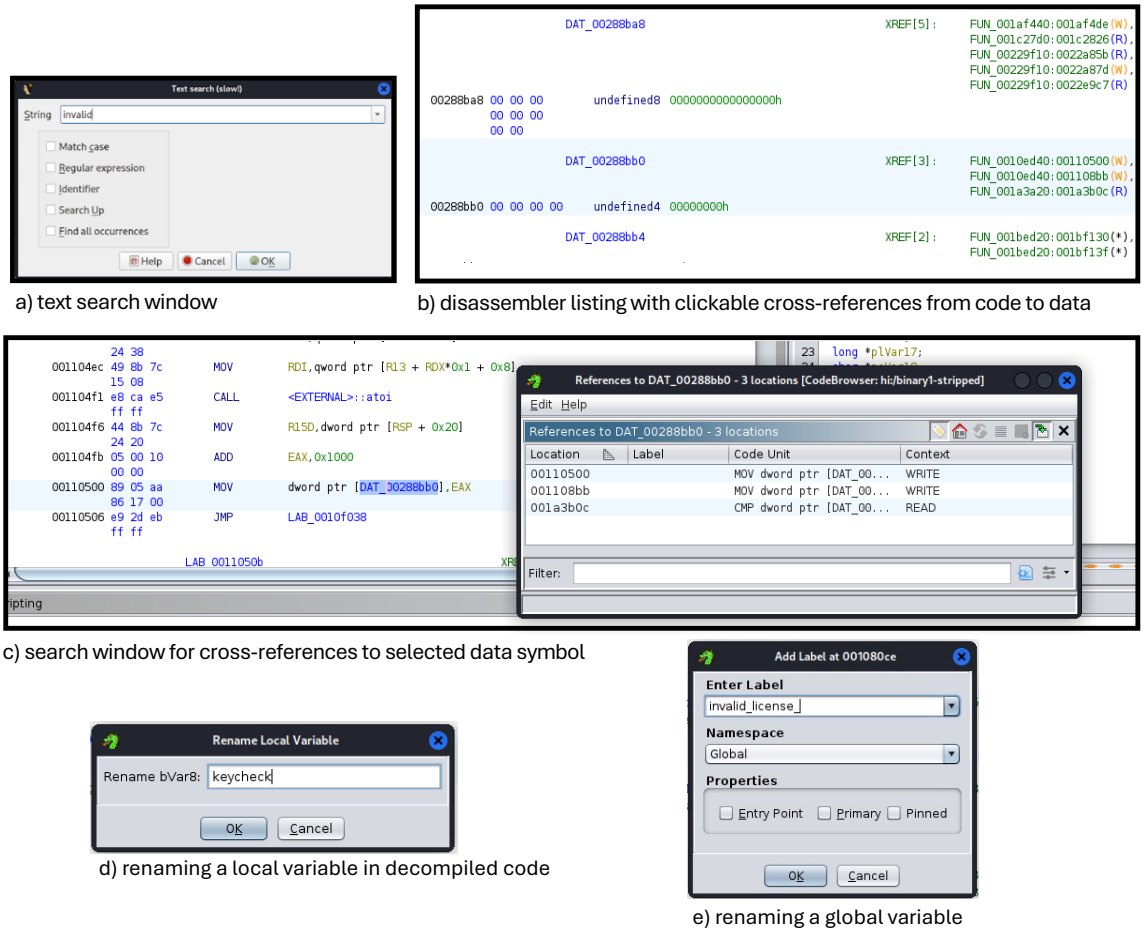


Figure 3: Screenshots of Binary Ninja illustrating how dynamically obtained data can be used to "optimize" the standard program representation in the form of Binary Ninja's high-level intermediate language (HLIL).



a) text search window

b) disassembler listing with clickable cross-references from code to data

c) search window for cross-references to selected data symbol

d) renaming a local variable in decompiled code

e) renaming a global variable

Figure 4: Screenshots of Ghidra functionality for searching, navigating, labeling, and renaming symbols.

CFG before we asked Binary Ninja to update its computed value sets with the dynamically obtained data from the KB.

However, according to that dynamic data, the currently selected instruction `mov rdx, qword [rsi+rax]` only reads zero values. When we import that information into Binary Ninja using the dedicated menu option we added through its customization APIs, Binary Ninja recomputes its value sets, discovers that one of the paths becomes unrealizable, makes it disappear from the constructed and shown CFG, and removes code that has now become dead (i.e., operations underlined in red in ④, resulting in the optimized HLIL CFG ⑤ on the right of Figure 3. In our opinion, this CFG matches the mental model of the code that a RE would have built by combining their static and dynamic information. Note that in the assembler CFG, Binary Ninja only shows the original, unoptimized assembly code. That is a limitation of Binary Ninja. However, the `{0x1}` after `je .skip_move` does indicate that Binary Ninja has deduced that this conditional branch will always be taken. So while the result of the code optimization is not visible, the result of the updated VSA is.

B Navigation, Searching, and Renaming

Figure 4 a–c show some example screenshots of search and navigation functionality in the Ghidra interactive disassembler/decompiler. It is the use of these types of functionalities that the tools discussed in Section 5 can extract from data collected during REing experiments with human REs. Similarly, Figure 4 d–e shows some example screenshots of functionality to give (new) names to artifacts in disassembled and decompiled code. These new names are a form of comprehension results, and the renaming activities indicate that the users assume they have obtained useful knowledge. Also the use of these functionalities can be extracted as discussed in Section 5.