TRINITY: An IDE for The Matrix

Jeroen van den Bos
Netherlands Forensic Institute (NFI)
The Hague, The Netherlands
Email: jeroen@infuse.org

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands
Email: storm@cwi.nl

***Abstract*—Digital forensics software often has to be changed to cope with new variants and versions of file formats. Developers reverse engineer the actual files, and then change the source code of the analysis tools. This process is error-prone and time consuming because the relation between the newly encountered data and how the source code must be changed is implicit. TRINITY is an integrated debugging environment which makes this relation explicit using the DERRIC DSL for describing file formats. TRINITY consists of three simultaneous views: 1) the runtime state of an analysis, 2) a hexview of the actual data, and 3) the file format description. Cross-view traceability links allow developers to better understand how the file format description should be modified. TRINITY aims to make the process of adapting digital forensics software more effective and efficient.**

I. BACKGROUND

A. Maintenance Challenges in Digital Forensics

The storage capacity of digital devices continues to grow. Forensic software is currently required to analyze data in the terabyte range in very short time frames. This requires perfective maintenance to optimize and tune analysis tools. At the same time, corrective maintenance has to be performed when new variants and versions of file formats are encountered. Most of these variants are non-standard, so standards documents cannot be consulted for the required changes. Moreover, the data is often created by proprietary firmware (e.g., of digital cameras) or other types of closed-source applications (e.g., word processors, photo-editing software). As a result, the source code is generally unavailable for inspection.

Corrective maintenance then boils down to reverse engineering the file format variant based on the binary data itself. This process is quite cumbersome, since the structure of the data is not a first class citizen in general purpose programming languages. In hand-coded file processing software, the layout of a binary file format like PNG [1], for instance, is encoded in complex control-flow and (interdependent) data structures. This means that debugging requires ad hoc decoding of values, inspection of input data to check dependencies between values and manually tracking structural layout and ordering.

Besides time consuming, these steps also tend to be error-prone. For example, an off-by-one error in an offset calculation causes a wrong value to be used, but also shifts interpretation of all consecutive values and their dependencies. Such small errors are hard to catch since there are no explicit links between the input data and how the code interprets it.

When adapting existing implementations of file processing software, interactive debuggers can be used, but they are agnostic to the domain-specific aspects of file formats. Furthermore, each file format may have its own conventions such as whether length fields include or exclude marker values, and whether indices are 0- or 1-based. As a result, reverse engineers have to mentally translate the information that is presented to them.

TRINITY is available at `http://github.com/jvdb/trinity`.

TRINITY is an IDE for reverse engineering binary data which automates a significant portion of this translation. By maintaining semantic links between data, runtime state and code, it becomes possible to *debug the data*, instead of just the code. The key enabler for this is representing file format structure at a higher level of abstraction. DERRIC is a domain-specific language (DSL) that precisely does that [2].

B. Declarative File Format Descriptions

DERRIC is a domain-specific language to declaratively describe binary file formats. It allows the definition of the components of a file format (called "structures"), their sequential arrangement, and the possible dependencies between elements. For instance, a file format description may contain structure definitions for headers, footers and data blocks. These structures are arranged sequentially according to a (regular) grammar, capturing the layout of a file format. An example of a dependency is when the length of a certain sequence of bytes is constrained by the value of certain bytes elsewhere in the file. DERRIC provides a configurable language for expressing these and other aspects of file formats.

A DERRIC description is divided in two main sections. The first part of a DERRIC description is the sequence section, which consists of a regular expression capturing the sequential layout of a file format. For instance, the following example presents an abridged version of the layout of PNG (where ellipses indicate omitted details):

```
sequence
  Signature IHDR
    (...)* PLTE? (...)* IDAT IDAT* (...)*
  bBPn? IEND?
```

The regular operators $*$ and ? have the usual meaning of repetition and optionality. The identifiers (e.g., `Signature`, `IHDR`, etc.) refer to specific components of PNG. These structures are described in the second part of a DERRIC description. As an example, the following snippet describes the `IEND` structure:

```
IEND {
  length: 0 size 4;
  chunktype: "IEND";
  crc: 0xAE, 0x42, 0x60, 0x82
}
```

This declaration states that the `IEND` structure consists of a length field of 4 bytes (containing zeros), followed by the (ASCII encoded) string "IEND", and terminated by a CRC code consisting of 4 constant values. To factor out common fields in structure definitions, DERRIC allows structures to inherit from other structures. For instance, in PNG, most structures inherit from an abstract `Chunk` structure which declares common fields for length, type, data and CRC check; such fields can be overridden if needed.

A DERRIC description is input to the DERRIC compiler which generates executable *validators*. A validator tries to match binary input streams against the file format definition captured in DERRIC. One application of these validators is *file carving*: the process of recovering possibly damaged or fragmented files from storage devices [3], [4]. Previous research has shown that the generated validators perform well, both in terms of recovered files and runtime speed [2], and that DERRIC descriptions can be automatically transformed to improve runtime performance [5].

The benefits of DERRIC are only fully realized, however, if the file format description can be considered correct. If files are encountered that are not recognized, there are two possibilities:

- The binary data is not an instance of the file format we are looking for, or the data is corrupted. In other words, the data is at fault.
- The file format description is incorrect and has to be changed to cope with this specific variation of the file format.

Note that these situations may overlap. In fact, it is quite common to relax a file format description to trade some precision for a higher recall. Nevertheless, in both cases the question remains: how to find out if a description should be adapted to the new situation? And if so, how should the description be changed? TRINITY helps to answer such questions by providing debugger functionality at the level of DERRIC itself. This way, both the data and the runtime state of an analysis can be interpreted in terms of the sequential layout and the structures and fields of the file format.

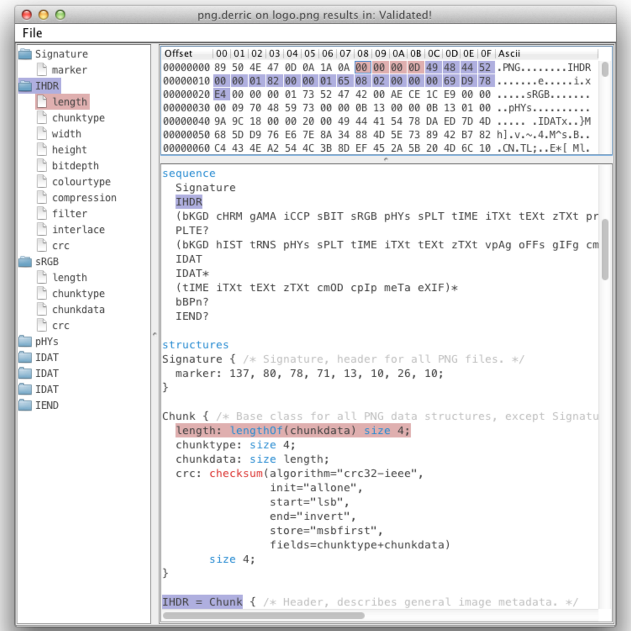
II. TRINITY

A. Integrated Data Debugging

TRINITY is an IDE which aims to leverage the domain-specific information contained in DERRIC descriptions to bring integrated data debugging support to the process of reverse engineering binary file formats. A screen shot of TRINITY is shown in Figure 1. The IDE consists of three synchronized views:

- **Data**: A hexview showing the input data (top right).
- **State**: An outline view of the runtime state, with root nodes for structures and child nodes for fields (left column).
- **Code**: A syntax-highlighting editor for showing a DERRIC description (bottom right).

The user can navigate between views using hyper links which connect all three views. For instance, after selecting the



```
Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F Ascii
00000000 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52 .PNG........IHDR
00000010 00 00 01 82 00 00 01 65 08 02 00 00 00 69 D9 78 .......e.....i.x
00000020 E4 00 00 00 01 73 52 47 42 00 AE CE 1C E9 00 00 .....sRGB.......
00000030 00 09 70 48 59 73 00 00 0B 13 00 00 0B 13 01 00 ..pHYs..........
00000040 9A 9C 18 00 00 20 00 49 44 41 54 78 DA ED 7D 4D ..... .IDATx..}M
00000050 68 5D D9 76 E6 7E 8A 34 88 4D 5E 73 89 42 B7 82 h].v.~.4.M^s.B..
00000060 C4 43 4E A2 54 4C 3B 8D EF 45 2A 5B 20 4D 6C 10 .CN.TL;..E*[ Ml.
```

```
sequence
  Signature
  IHDR
  (bKGD cHRM gAMA iCCP sBIT sRGB pHYs sPLT tIME iTXt tEXt zTXt pr
  PLTE?
  (bKGD hIST tRNS pHYs sPLT tIME iTXt tEXt zTXt vpAg oFFs gIFg cm
  IDAT
  IDAT*
  (tIME iTXt tEXt zTXt cmOD cpIp meTa eXIF)*
  bBPn?
  IEND?

structures
Signature { /* Signature, header for all PNG files. */
  marker: 137, 80, 78, 71, 13, 10, 26, 10;
}

Chunk { /* Base class for all PNG data structures, except Signatu
  length: lengthOf(chunkdata) size 4;
  chunktype: size 4;
  chunkdata: size length;
  crc: checksum(algorithm="crc32-ieee",
               init="allone",
               start="lsb",
               end="invert",
               store="msbfirst",
               fields=chunktype+chunkdata)
       size 4;
}

IHDR = Chunk { /* Header, describes general image metadata. */
```

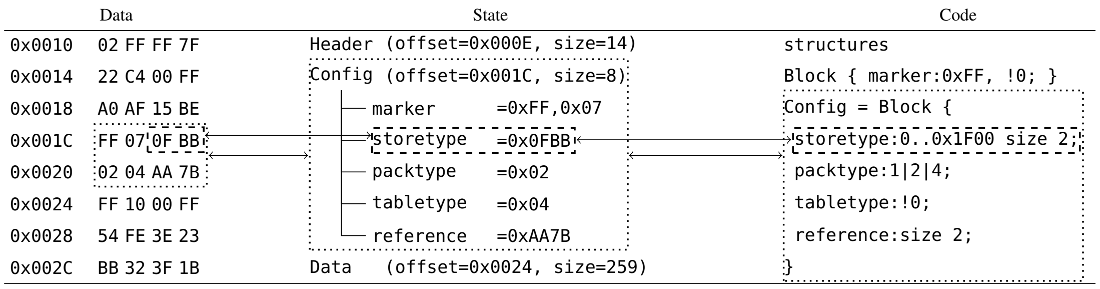
Fig. 1: Screenshot of TRINITY used on a PNG example file.

byte at offset 8 in the Data view at the top right, the contextual structure and field of this byte are highlighted. Similarly, the `IHDR` structure and its `length` field are highlighted in the State view on the left, which provides the dynamic execution context to this byte. In the Code view at bottom right, the `IHDR` structure is highlighted in both the **`sequence`** and **`structures`** sections. Finally, the `length` field is highlighted in the Code view as well, where it is defined not directly in the `IHDR` structure, but in the `Chunk` structure it inherits from.

It is also possible to go the other way. For instance, clicking on a field in the code view will highlight all the bytes in the input stream that have been successfully matched using that very field. Similarly, clicking on an element in the sequence section highlights all bytes in the input stream captured by that syntactic element. Because syntactic elements in the sequence may occur multiple times (through the use of the regular operator $*$), clicking on a source element may highlight multiple parts of the input data.

Figure 2 illustrates the relationships between the three views in more detail. On the left (Data) is a hexview of the input data (between offsets 16 (`0x0010`) and 48 (`0x002C` `+ 4`). In the center (State) the trace of interpreting the input data (showing matches for structures named `Header`, `Config` and `Data`, of which only `Config` is expanded and showing its fields). On the right (Code) the text editor view of the DERRIC description (showing the definition of the `Config` structure). In all three views, the dotted line marks the `Config` structure and the dashed line its `storetype` field.

By making the links between data, runtime state and code explicit, TRINITY simplifies the reverse engineering and maintenance tasks in dealing with binary file formats. The developer can interactively explore the original file format



```
         Data                    State                                  Code
0x0010   02 FF FF 7F    Header (offset=0x000E, size=14)          structures
0x0014   22 C4 00 FF    Config (offset=0x001C, size=8)           Block { marker:0xFF, !0; }
0x0018   A0 AF 15 BE      ├── marker      =0xFF,0x07             Config = Block {
0x001C   FF 07 0F BB      ├── storetype   =0x0FBB                 storetype:0..0x1F00 size 2;
0x0020   02 04 AA 7B      ├── packtype    =0x02                   packtype:1|2|4;
0x0024   FF 10 00 FF      ├── tabletype   =0x04                   tabletype:!0;
0x0028   54 FE 3E 23      └── reference   =0xAA7B                 reference:size 2;
0x002C   BB 32 3F 1B    Data   (offset=0x0024, size=259)         }
```

Fig. 2: The relationship between Data view (left, hexview), State view (center, outline) and Code view (right, text editor).

description in DERRIC directly in the context of the actual bytes in the input data. Below we describe how TRINITY can be used in digital forensics practice.

B. A File Format Reverse Engineering Scenario

The design of TRINITY is informed by more than a decade of experience in reverse engineering file formats. Additionally, in previous research we have performed an experiment which studied corrective maintenance of DERRIC descriptions [6][1] by executing evolution scenarios. These scenarios for "fixing" the descriptions all represent typical cases where TRINITY could be used. In fact, the research of [6] would have been much less time consuming if TRINITY had been available at the time, as most of the effort consisted of relating error locations in binary data to source locations in DERRIC.

The use of TRINITY starts when a file is encountered that is expected to validate, but fails to do so. The following steps describe the expected work flow using TRINITY:

1) Initial Run: The file and the DERRIC description of its expected file format are loaded into TRINITY and the interpreter halts at the first byte where validation fails. The file's contents is shown in the Data view, the DERRIC description in the Code view and the generated trace after an initial run in the State view.

2) Locate Area of Interest: The user clicks on the last data structure listed in the trace, automatically showing the relevant child nodes. The Data view is automatically scrolled to the corresponding bytes. The cursor in the Code view is positioned on the structure where validation failed.

3) Inspect Structure: The user clicks the last field below the structure in the trace. This keeps the existing highlighting but adds additional ones of the fields' bytes in the Data view and its description in the Code view.

4) Make Corrections: Based on whether that field is the source of the validation error, the user will either make a modification or move up to the previous field, backtracking until a field or structure is encountered which accounts for the failure. Finally, the validation is rerun, and the process repeats if there are (new) failures.

III. IMPLEMENTATION

DERRIC is implemented as an external DSL in the metaprogramming language Rascal [7]. Rascal provides built-in grammars for describing syntax, primitives for analyzing and transforming source code, and provides hooks into the Eclipse IDE to obtain editor services (e.g., syntax coloring, outlining, hyperlinking etc.).

[1]The changes can be reviewed online at http://github.com/jvdb/derric-eval/.

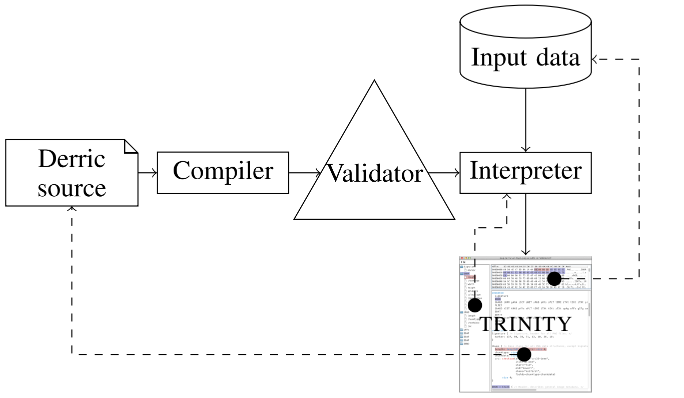


Fig. 3: The TRINITY architecture. The dashed arrows indicate the information sources of the three views in the IDE.

The DERRIC compiler operates in three steps. First the DERRIC description is desugared (e.g., flattening inheritance, constant propagation). Second, a DERRIC description is transformed to an intermediate representation called Validator, which is an imperative but platform-independent model of the final validator. Finally, the Validator model resulting from the previous step is transformed to Java source code.

An overview of the architecture of TRINITY is shown in Figure 3. TRINITY reuses the front-end part of the DERRIC compiler, up to and including the transformation to the Validator model. Instead of generating Java code however, the Java foreign-function interface of Rascal is used to build an in-memory model in Java of the Validator. Following the Interpreter design pattern, the classes representing the model contain evaluation methods to execute the validator. This interpreter is then hooked up to the TRINITY IDE.

To realize the fine-grained cross-linking of views in TRINITY, origin tracking is used [8]. This means that the original source locations of syntactic elements in a DERRIC description are maintained throughout all phases of the compiler and interpreter. The DERRIC parser generated by Rascal initially annotates the parse tree with such origins. During desugaring and the transformation to the Validator model, the origins are propagated. Finally, the in-memory model in Java is decorated

$$\begin{tikzcd}
 & \text{Data} \arrow[dl, shift right] \arrow[dr, shift left] & \\
\text{State} \arrow[ur, shift right] \arrow[rr, shift right] & & \text{Code} \arrow[ul, shift left] \arrow[ll, shift right]
\end{tikzcd}$$

Fig. 4: The trinity of debugging in TRINITY

so that, when the interpreter is stopped, the TRINITY runtime environment knows where in the DERRIC description execution is taking place. The same technique is used to maintain a mapping from the runtime state (i.e. the values of the matched structures and fields), to the source code, and from the source code to the data.

IV. RELATED WORK

The key idea of TRINITY is to integrate the input data into the activity of debugging and to provide bidirectional cross-links among code, state and data. Moreover, the integration is domain-specific: DERRIC descriptions capture file formats at a level that can be understood by forensic investigators. In TRINITY this understandability extends to the data and the runtime state of the validator. As a result, TRINITY provides debugging for reverse engineering file formats at a higher level of abstraction.

Using TRINITY the user can navigate from the source code to the data and vice versa, but also from the runtime state to the data and vice versa, and finally, it is possible to go from the data to the runtime state and the source code. We have depicted these 6 types of cross links in Figure 4. Traditional debuggers, on the other hand, provide only two of such links: 1) from the runtime state (e.g., stack trace) to the source code, and 2) from the data to the code (e.g., from a variables view to declaration sites). Although specialized visualizations for general purpose debuggers are quite common (e.g., [9], [10]), these do not provide the same level of integration as TRINITY.

TRINITY is most related to domain-specific debuggers in other domains. For instance, ANTLRWorks [11] is an IDE which provides support for debugging ANTLR grammars. The generated parsers communicate with the IDE during their execution, allowing the user to replay its actions and inspect the input data, grammar and parse tree afterwards. Similar tools exist for debugging regular expressions. A recent example is Debuggex [12], which features coloring of the (matched) input data, and visualization of the finite-state automaton.

V. CONCLUSION AND FUTURE WORK

Reverse engineering binary file formats is a time-consuming and error-prone activity. One of the reasons is that the relation between the structure of the data and how software processes that data is obscured by low-level implementation details and has to be mentally reconstructed. In this paper we have presented TRINITY, an IDE that brings integrated data debugging support to the DERRIC IDE for file format description. It consists of three views, which display the input data, the runtime state of a file format validator and the DERRIC source code respectively. Each view is related to the other. Clicking in any of the views highlights corresponding elements in the others. If a file fails to validate, the three integrated views allow the developer to assess the situation: why does validation fail? What changes are needed to the file format description? TRINITY aims to reduce the effort of performing corrective maintenance of digital forensics software.

There are ample opportunities for further improving TRINITY. For instance, the way elements are highlighted in the different views is mostly syntactic. One extension would be to add more semantics to the visualization. For instance, clicking a field that has a dependency on another field in its length or content specification, could also highlight the bytes that were captured by those dependency fields. Conversely, clicking on a byte in the data view could also trigger highlighting of all expressions affected by it. Such data flow visualization could further increase understanding of what happens at runtime and help diagnosing failures.

Another direction for further work is increasing the "liveness" of TRINITY [13]. Currently, TRINITY allows the dynamic inspection of state and data. However, changes to the DERRIC description still requires a full rerun of the validator. The potential benefits presented by TRINITY could be increased further by instantly reflecting a change to the format description in the other views. One way to approach this is to incrementally update the runtime state of the interpreter based on the changes to the code (see, e.g., [14]).

Finally, we plan to perform a user study to evaluate to what extent TRINITY helps to improve the maintenance of DERRIC descriptions. The evolution scenarios obtained in [6] can provide a starting point for the maintenance tasks to set up this experiment.

REFERENCES

[1] W3C, "PNG Specification," 2003, http://www.w3.org/TR/PNG/.

[2] J. van den Bos and T. van der Storm, "Bringing Domain-Specific Languages to Digital Forensics," in *ICSE'11*. ACM, 2011, pp. 671–680.

[3] M. I. Cohen, "Advanced Carving Techniques," *Digital Investigation*, vol. 4, no. 3-4, pp. 119–128, 2007.

[4] A. Pal and N. Memon, "The Evolution of File Carving," *Signal Processing Magazine*, vol. 26, no. 2, pp. 59–71, 2009.

[5] J. van den Bos and T. van der Storm, "Domain-Specific Optimization in Digital Forensics," in *ICMT'12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 121–136.

[6] ——, "A Case Study in Evidence-Based DSL Evolution," in *ECMFA'13*, ser. LNCS, vol. 7949. Springer, 2013, pp. 207–219.

[7] P. Klint, T. van der Storm, and J. Vinju, "Rascal: A Domain Specific Language for Source Code Analysis and Manipulation," in *SCAM'09*. IEEE, 2009, pp. 168–177.

[8] A. v. Deursen, P. Klint, and F. Tip, "Origin tracking," *Journal of Symbolic Computation*, vol. 15, pp. 523–545, 1993.

[9] A. Zeller and D. Lütkehaus, "DDD - A Free Graphical Front-End for UNIX Debuggers," *SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[10] Hex Rays, "IDA," https://www.hex-rays.com/products/ida/index.shtml.

[11] J. Bovet and T. Parr, "ANTLRWorks: an ANTLR grammar development environment," *Softw., Pract. Exper.*, vol. 38, no. 12, pp. 1305–1332, 2008.

[12] S. Toarca, "Debuggex," http://www.debuggex.com/.

[13] H. Lieberman and C. Fry, "Bridging the Gulf between Code and Behavior in Programming," in *CHI'95*. ACM, 1995, pp. 480–486.

[14] T. van der Storm, "Semantic deltas for live DSL environments," in *LIVE'13*. IEEE, 2013, pp. 35–38.