Abstract

GTIRB is an intermediate representation for binary analysis and transformation tools including disassemblers, lifters, analyzers, rewriters, and pretty-printers. GTIRB is designed to enable communication between tools in a format that ensures the basic information necessary for analysis and rewriting is provided while making no further assumptions about domain (e.g., malware vs. cleanware, or PE vs. ELF) or semantic interpretation (functional vs. operational semantics). This design supports the goals of (1) encouraging tool modularization and re-use allowing researchers and developers to focus on a single aspect of binary analysis and rewriting without committing to any single tool chain and (2) easing communication and comparison between tools.

1 Introduction

Software is essential to the functioning of modern societies. It follows that software analysis, hardening, and rewriting are essential to the secure and efficient functioning of society. Unfortunately, and quite frequently, software is only available in binary form, whether as dependencies of active software projects, firmware and applications distributed without source access, or simply old software. Both analyzing and rewriting require first lifting software to an initial intermediate representation (IR). Binary analysis frameworks typically develop and use their own internal IR [9, 3, 12, 1, 7, 18, 14], in some cases IRs are borrowed from other tools such as dynamic analysis tools [20] or compiler infrastructure [11, 13]. The representations used by these tools typically specify the representation of instruction semantics, which in turn often dictates the methods of analysis and the programming languages used by their clients. These IRs are typically not portable between tools and projects.

GTIRB is intended to facilitate communication between tools for binary analysis and transformation. GTIRB is released as open-source software [311x415] with a high quality disassembler, Ddisasm [311x400], capable of lifting COTS binaries to GTIRB [10]. To ensure applicability across domains, GTIRB’s structural requirements are minimal. To ensure interoperability between tools regardless of their instruction semantics, GTIRB does not represent instructions; instead the raw machine-code bytes are stored in the IR. To avoid language lock-in, GTIRB is serialized using Protobuf [1], an efficient multi-language serialization library. By allowing communication between tools and the modularization of monolithic tools, we hope that GTIRB will enable greater re-use of components across the binary analysis and rewriting community.

The LLVM project [15] demonstrates the huge benefit a well designed IR can have on a research community. LLVM allows compiler researchers to more easily leverage each other’s work and focus on the problems specific to their own research. This has led to dramatic uptake of LLVM and Clang across a number of research areas as well as in industry. GTIRB seeks to recreate LLVM’s success for the binary analysis and transformation community.

arXiv:1907.02859v1 [cs.PL] 2 Jul 2019
2 Related Work

IDA IDA Pro [2] is the industry leading binary analysis and reverse engineering platform. It provides disassembly, decompilation, and an interactive environment for navigating binary programs. IDA is extensible through a plugin API, and there are a sizeable number of open source plugins that have been developed by the community. The disassembly produced by IDA is primarily intended to support manual review and is not intended to support reassembly.

Ghidra Ghidra [3], recently released by the national security agency (NSA), is a reverse engineering framework providing a graphical user interface based on Eclipse. Like IDA, Ghidra is extensible, supporting scripts and plugins. Ghidra is also primarily intended to support manual analysis and the disassembly and decompilation provided by Ghidra are not primarily intended for reassembly or recompilation.

Angr Angr [20] is currently the most widely used binary rewriting platform. Angr is a platform that is used via a suite of Python 3 libraries. The platform provides functionality for disassembly, analysis, and symbolic execution. Ramblr [22], the reassembleable disassembler, which currently boasts the best published results, is part of the Angr framework. Angr uses the Vex instruction representation from Valgrind [17] to represent instructions.

BAP The CMU’s Binary Analysis Platform (BAP) [7] lifts binaries to its Binary Intermediate Language (BIL) using tooling based on either IDA Pro or LLVM. BAP plugins may then be written to use the BIL representation of the software.

Uroboros Uroboros [23] was the first tool to focus directly on generating reassembleable assembly. Uroboros directly outputs text assembler code. Rewriting is done by compiling plugins into Uroboros to modify simple instruction data structures.

Multiverse Multiverse [6] is a static binary rewriter which does not use heuristics but reassembles all possible disassemblies. The Capstone disassembler is used, and a simple Python API may be used to add instrumentation.

LLVM LLVM [15] provides an IR that is a popular target for language front-ends, most notably the Clang C/C++ front end. The rich ecosystem of optimization and analysis tools implemented over-top of LLVM make it an attractive target. There are a number of projects seeking to lift binary software to LLVM, most notably McSema [8] and Second-Write [21]. Unfortunately LLVM is a difficult target for binary lifting given the strongly typed memory model, which forces very difficult analysis decisions before the IR may even be constructed.

3 Design of GTIRB

An instance of GTIRB is a single data structure organized as shown in Figure 1. Every element of Figure 1 is tagged as either “(1)” or “(N)” indicating there is only one or possibly many instances of the element respectively.

3.1 Core Structures

At the top level of every GTIRB instance is a single IR element. This IR holds multiple Modules. Each module corresponds to a single compilation unit, e.g.
an executable or a shared library. A single GTIRB IR could represent a binary executable and all of the libraries it uses dynamically, each as a separate module. The two main portions of each module are the Blocks and the DataObjects, which represent the code and data of the module respectively. Both the Blocks and the DataObjects store their contents as regions of raw bytes in the single ImageByteMap associated with their module. The ImageByteMap is a sparse vector of bytes holding the raw contents of the module including all code and data. The ranges of both Blocks and DataObjects may overlap arbitrarily with the ranges of other Blocks and DataObjects.

GTIRB does not explicitly indicate the interpretation of the bytes in blocks or data objects. While blocks are notionally intended to represent basic blocks of instructions, a decoder is required to extract individual instructions from a code block (see §3.3.1). The interpretation of the bytes in a data object depends on the program being analyzed. The deduced type for a data object may be stored in an auxiliary data table (see §3.2).

GTIRB imposes an additional level of structure on code. The IPCFG is a single graph covering all code in the module (see §3.3.2) in which each node is a block and each edge connects two blocks (see §3.3.4). Edges between code blocks represent control flow in the IPCFG.

Symbols and SymbolicExpressions are explicitly represented by GTIRB. These provide symbolization information for symbols in code and data blocks. In the case of code blocks they indicate which operands are symbolic to specific instructions and, in the case of data, which data is symbolic. In both cases they hold a pointer to a block or data object and an offset into the byte contents. This allows precise location of the affected region of data or portion of a decoded instruction, while remaining agnostic to the instruction representation used for code blocks. The symbolization information required by GTIRB is sufficient to enable the contents of the binary to be reorganized in memory while updating all cross references to accommodate binary rewriting.

Finally, the Sections in GTIRB are used to store information on the loadable sections of the module. Every element of GTIRB, namely: Modules, Symbols, SymbolicExpressions, DataObjects, Blocks, Edges, and Sections have a universally unique identifier (UUID). UUIDs allow both first-class IR components and AuxData tables to reference other elements of the IR in a manner that is robust to rewriting. E.g., Symbols use UUIDs to reference blocks. Note that reference by address in the original binary would not be robust to rewriting as new entities could not be added to the IR without synthesizing fake addresses.

GTIRB may be serialized using Google’s protobuf [1], making it possible to efficiently read and write GTIRB from any language with Protobuf support. Currently there are custom GTIRB libraries for C++ and Python that provide more ergonomic and efficient APIs than the default Protobuf APIs.

### 3.2 Auxiliary Data Tables

The core GTIRB data structure described in §3.1 is intentionally very sparse. Even very generally useful information, e.g. the concept of functions, is not included by default in GTIRB because its use may not be universal, e.g. malware or hand-written assembler may not have functions. One of the core purposes of GTIRB is to communicate analysis results between tools but the only analyses explicitly representable in the core GTIRB structure are symbolization, CFG, and code vs. data. Much of the information of any instance of GTIRB is intended to be communicated not in the core required structures, but instead via auxiliary data (i.e., AuxData) tables. These tables are extensible and may be used to store maps and vectors of basic GTIRB types or arbitrary data in a portable way. AuxData tables make heavy use of UUIDs to reference elements of the core GTIRB IR.

For very common information such as function boundaries there are “sanctioned” AuxData table schemas. By standarizing types for commonly used tables we hope to ensure compatibility between tools. We anticipate adding new schemas to the list of sanctioned AuxData tables as they become widely used. We list the current sanctioned AuxData schemas in Table 1.

| Table 1 | The sanctioned tables in Table 1 have the following meanings. |
Table 1: Sanctioned schemas of commonly useful AuxData tables. Shown as C/C++ types.

| Label            | Type                                                                 |
|------------------|----------------------------------------------------------------------|
| functionBlocks   | std::map<gtirb::UUID, std::set<gtirb::UUID>>                        |
| functionEntries  | std::map<gtirb::UUID, std::set<gtirb::UUID>>                        |
| types            | std::map<gtirb::UUID, std::string>                                  |
| alignment        | std::map<gtirb::UUID, uint64_t>                                     |
| comments         | std::map<gtirb::Offset, std::string>                                |
| symbolForwarding | std::map<gtirb::Symbol,gtirb::Symbol>                              |

functionBlocks Along with functionEntries this table identifies function boundaries. A function is stored as a set of code blocks. Storage as a set instead of a region of memory ensures robustness to modification of the IR and permits the representation of non-contiguous functions.

functionEntries Stores the set of blocks used as entry points to a function. Representation of multiple-entry functions is supported.

types The type of a DataObject. The type of the data is expressed as a string containing a valid C++ type specifier.

alignment The preferred alignment of a Block or DataObject in memory (see §3.3.7).

comments Supports the storage of arbitrary comments stored as strings, which reference particular offsets within blocks (e.g., an instruction in a code block or a particular point within a data element).

symbolForwarding This table redirects one symbol to another. This is useful to resolve indirections related to dynamic linking. For example, it connects symbols pointing to PLT entries to the function symbols called in such PLT entries. It also resolves indirect references via the GOT table.

3.3 Design Decisions

Many decisions were made in the design of GTIRB. These were motivated by (i) our experience in the development and use of tools for binary analysis and rewriting, (ii) a desire to maximize generality and flexibility, and (iii) a desire for simplicity and orthogonal elementary concepts when possible. In this section we discuss some of the potentially more surprising decisions we made.

3.3.1 Instruction Storage

The most frequent misconception about GTIRB is that it is an intermediate language (IL) for representing the semantics of assembler instructions in the same way that BAP’s BIL [3], Angr’s Vex [4] or Ghidra’s P-code are ILs. GTIRB represents the higher-level structure of the binary. These structures are often the result of sophisticated analyses (e.g., those performed by our front end, Ddisasm).

For instruction representation GTIRB uses the most general and efficient representation available, and possibly some of the most over-engineered serialization encoding in human history, the raw machine code bytes. The users of GTIRB may read/write these bytes using the decoder/encoder of their preferred IL (e.g., BIL, Vex, P-code) or using the high quality open-source Capstone [5]/Keystone [6] libraries.

This decision has the benefits of universality and memory efficiency. Often AST representations of instructions can incur very large space overheads of many times the space required by the machine code bytes. This overhead is often the limiting factor when analyzing large binaries, or collections of models. Decoding and encoding machine-code bytes as needed
permits fast access to an extremely efficient representation of the code. The universality of machine-code bytes ensures that the core mission of interoperability is not compromised, and even permits useful flexibility within a single project or framework.

The main drawback to this decision is that GTIRB does not provide instruction semantics. However, there are already many powerful tools in this space, such as those referenced at the beginning of this section, as well as emerging standards. In our experience using GTIRB with our own custom instruction semantics, the access patterns required by machine code bytes are manageable and well worth the benefits.

3.3.2 CFG vs. IPCFG

The use of an IPCFG instead of a typical CFG with functions is a result of the choice to not have first-class functions (§3.3.3). By dispensing with the intermediate decomposition of the CFG into procedures an IPCFG is simpler to build and simpler to use in many cases. Importantly it ensures that subsequent analyses are only dependent on detangling the often tricky edge cases of function boundary identification when those analyses explicitly require this information. Forcing the encoding of functions into a CFG would make this an implicit potential source of error for any analysis using the CFG, even those which don’t require function information.

The IPCFG also opens the door to non-standard code representations, such as dispensing with the notion of basic blocks and instead representing the code section as a graph of single instructions joined by control flow edges—as done by SEI’s Pharos [13]. (This is easily represented in GTIRB using single-instruction code blocks.)

3.3.3 Second-class functions

Functions are not essential to a functioning binary, e.g., malware and hand-written assembler may dispense with the function abstraction. Even in compiled code function boundary identification is a difficult problem and an active research area [16] [5] [19]. However, many static analyses and transformations require function boundaries to work. Thus, we allow for the representation of functions as sets of basic blocks (and sets of entry points) in AuxData tables. This also simplifies the CFG representation.

3.3.4 Block types and Edge types

Blocks represent a range of bytes in their module’s ImageByteMap that are interpreted as code. (Bytes interpreted as data are represented by DataObjects.) The range of addresses covered by each Block may include a number of distinct instructions. Although GTIRB does not represent these individual instructions explicitly, the implication is that control flow fall through from one instruction to the next within a single Block. That is, non-local control flow such as branches, calls, and returns occurs only at the end of a Block. GTIRB does not require that each Block represent a basic block, although we expect that to be the most common usage. As mentioned above (§3.3.2), single-instruction blocks also meet this minimal requirement and may be comfortably represented in GTIRB.

Blocks constitute the nodes of the GTIRB IPCFG; the information about the local or non-local control flow between blocks is encoded as labeled edges. Edges can be labeled as conditional or unconditional, direct or indirect, and with the type of control flow between blocks. Supported types include branches, calls, returns, system calls, system call returns, and fallthrough. In combination, these allow one outgoing edge from a block to be labeled as a direct branch taken when a condition is true while another edge from the same block may be labeled as falling through to a subsequent block when the condition is false.

3.3.5 Extra-module edges

The IPCFG represents the control flow between blocks in a single module. To represent control flow between blocks in different modules, GTIRB uses proxies. A proxy block may be used as a node in the module’s IPCFG, but has no corresponding range of bytes. This allows representing calls to external functions, even when the library providing that function is not available for analysis.
For example, to represent a call to a function defined in another module, a client may insert a proxy into the IPCFG to represent the external function, then insert an edge between the calling block and the proxy. Similarly, if desired, a call from an external block can be represented by introducing a proxy to represent the caller and an edge from that proxy to the entry block of the called function.

3.3.6 Explicit Symbolization

Explicitly required symbolization information, as opposed to an optional symbolization auxiliary data table, is a result of our focus on supporting rewriting and the movement of code and data. Despite the structural requirement for symbolization information, it is still possible for a tool to populate a GTIRB instance while leaving the Symbols and SymbolicExpressions sections empty. Similarly the IPCFG could be left as a single block holding all of the code of a module, or as a series of disconnected blocks. While these extremes are not anticipated to be the common case, it is expected that most tools producing GTIRB from binaries will not produce perfect IPCFGs or symbolization information, and the corresponding GTIRB structure is intended to gracefully handle incomplete information.

3.3.7 Explicit Padding vs. Alignment

When dealing with compiler-generated padding between functions in the code section of a binary there are multiple valid representation options.

**Code** Padding regions are typically packed with nop instructions generated by the compiler to fill the space. As these are technically executable code one could represent the padding regions as code blocks that are simply disconnected from the remainder of the CFG.

**Padding** One could explicitly mark these regions as padding blocks that are not code blocks but are located in the code section. Marking this explicitly gives the user confidence that the blocks are not simply missed code but were actively identified as padding.

**None** The blocks could not be represented at all, but the appropriately inferred alignment directives could be placed on the subsequent code block.

GTIRB takes the “None” option of adding alignment directives to code blocks (see “alignment” in Table 1) instead of explicitly representing padding. This was done to avoid introducing useless disconnected nop-only blocks to the IPCFG (the “Code” option) and avoid adding heterogeneity of node types to the IPCFG with special padding blocks, which traversals would then have to handle (the “Padding” option).

4 Conclusion

GTIRB is an intermediate representation of the structure of binaries, indeed to facilitate communication between tools for binary analysis and transformation. An explicit design goal has been to enable flexibility and extensibility while providing a minimal core structure. This enables incremental lifting and analysis, since additional structure may be added in subsequent phases. It also encourages interoperability between tools written in many languages and on top of different analysis frameworks and semantics, through the medium of a language-agnostic serialized format. We hope that making GTIRB and our high-quality Disasm frontend open-source will stimulate a robust ecosystem of interoperable binary rewriting tools.

5 Acknowledgments

Many thanks to our colleagues at GrammaTech who contributed to the design and implementation of GTIRB especially; Brian Alliet, Abhishek Bhaskar, John Farrier, and Nathan Weston.

This material is based upon work supported by the Office of Naval Research under contract No. N68335-17-C-0700. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research.
References

[1] Protocol buffers. https://developers.google.com/protocol-buffers/.

[2] Hex-rays: The ida pro disassembler and debugger. https://www.hex-rays.com/products/idapro.

[3] National Security Agency. Ghidra, 2019. https://www.nsa.gov/resources/everyone/ghidra/.

[4] Cryptic Apps. Hopper. https://www.hopperapp.com/.

[5] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. Proceedings of USENIX Security 2014, 2014.

[6] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In NDSS, 01 2018.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, Computer Aided Verification, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] ARTEM DINABURG and ANDREW RUEF. Mcsema: Static translation of x86 instructions to llvm. In ReCon 2014 Conference, Montreal, Canada, 2014.

[9] Chris Eagle. The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler. No Starch Press, 2011.

[10] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. arXiv e-prints, page arXiv:1906.03969, Jun 2019.

[11] Galois Inc. Open source binary analysis tools. https://github.com/GaloisInc/macaw.

[12] Vector 35 Inc. Binary ninja: a new kind of reversing platform. https://binary.ninja/.

[13] Software Engineering Institute. Automated static analysis tools for binary programs. https://github.com/cmu-sei/pharos.

[14] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2r2: Building an efficient front-end for binary analysis. In Binary Analysis Research (BAR), 2019, 2019.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO ’04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pages 24–35, New York, NY, USA, 2016. ACM.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Programming Language Design and Implementation, pages 89–100, 2007.

[18] pancake. radare. https://www.radare.org/r/.

[19] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In 24th USENIX Security Symposium (USENIX Security 15), pages 611–626, 2015.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In 2016 IEEE Symposium on Security and Privacy (SP), pages 138–157, May 2016.
[21] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In Reverse Engineering (WCRE), 2013 20th Working Conference on, pages 52–61. IEEE, 2013.

[22] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In NDSS, 2017.

[23] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In 24th USENIX Security Symposium (USENIX Security 15), pages 627–642, Washington, D.C., 2015. USENIX Association.