Netlist Decompilation Workflow for Recovered Design Verification, Validation, and Assurance

Katie Liszewski
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
liszewski@battelle.org

Tim McDonley
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
mcdonley@battelle.org

Josh Delozier
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
delozier@battelle.org

Andrew Elliott
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
elliottas@battelle.org

Dylan Jones
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
jonesdt@battelle.org

Matt Sutter
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
sutter@battelle.org

Adam Kimura
Cyber Trust & Analytics
Battelle Memorial Institute
Columbus, OH USA
kimura@battelle.org

I. Introduction

Over the last few decades, the cost and difficulty of producing integrated circuits at ever shrinking node sizes has vastly increased, resulting in the manufacturing sector moving overseas. Using offshore foundries for chip fabrication, however, introduces new vulnerabilities into the design flow since there is little to no observability into the manufacturing process. At the same time, both design and optimization are becoming increasingly complex, particularly as SoC designs gain popularity. Common practices such as porting a design across node sizes and reusing cores at multiple area/performance tradeoffs further complicate assurance as layout specific features impede comparison.

Methods have been developed for conducting integrated circuit decomposition on fabricated chips [1][2][16] to extract the as-fabricated design files such as the GDSII layout or gate-level netlist. While mature netlist equivalency checking tools are included with any design flow, there is a lack of tools for performing deeper analyses on the extracted designs for the purposes of hardware assurance or design recovery from obsolete parts. To this end, there is a need for a tool to extract functionality from netlists at a higher abstraction level to reconstruct behavioral Register Transfer Level (RTL) code.

II. Leveraging Software Decompilation as an Analog to Hardware Decompilation

Software decompilation is a well-established technique that has been used since the 1960s to recover lost source code, verify code against design changes produced by the compiler, and support detection of malicious code. In seeking to recover RTL, these 80 years of expertise in reconstructing functionality are invaluable. We introduce the terminology of "hardware decompilation" and explore where software techniques are relevant, how existing netlist structure recovery techniques fit into the decompilation pipeline, and present new techniques that are unique to hardware decompilation.

The general phases of decompilation and organizational structure that this paper draws from, as outlined by Cristina Cifuentes [3], consists of three basic steps. In software decompilation, a parser lifts or converts low-level code into language-independent mnemonics called an intermediate language (IL), while maintaining design hierarchy using abstract syntax trees (ASTs) or similar hierarchical data structures. An analysis phase recovers control flow and data or register groupings by considering the software binary as a network graph. This phase can be augmented using well-established graph and information theoretic techniques including identification of known functions, referred to as signature identification. The final phase simply compiles the IL into the language of choice. **Figure 1** contrasts the software decompilation and proposed hardware decompilation flows.

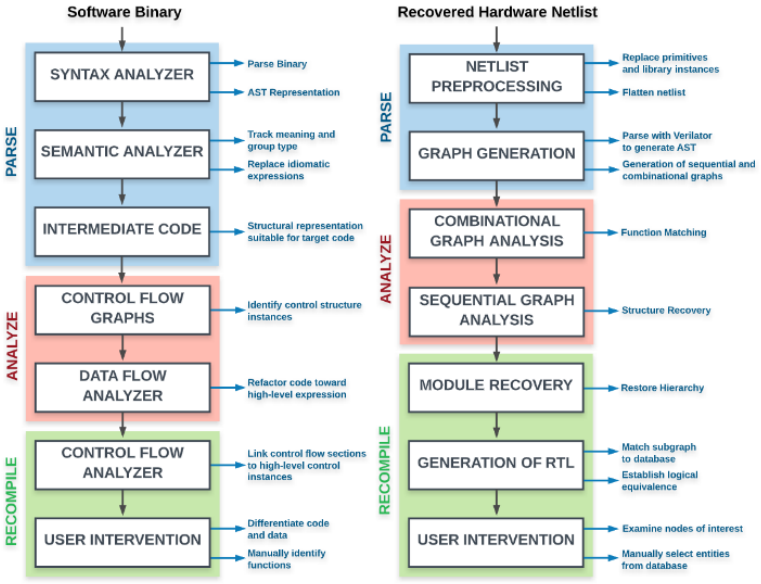


Figure 1. Traditional software decompilation (*left*) and proposed hardware decompilation for behavioral recovery of an extracted netlist (*right*)

DISTRIBUTION A. Approved for Public Release
Approval ID: 88ABW-2019-5298.

III. NETLIST DECOMPILATION

A. Netlist Preprocessing and Graph Generation

Both hardware and software decompilation begin with a format that is designed for machine use and not amenable to human interaction, literal circuitry or bitstreams for ASICs and FPGAs respectively and machine code for software. The software workflow starts with recognizing and decoding the binary file format. This forensics process is akin to decoding a bitstream into a netlist and has been studied by [13][14] and others. Recovering a netlist from an ASIC is much more difficult and error prone as it requires advanced material decomposition and imaging capabilities [17]. Decompilation aids in verifying a netlist was accurately recovered by producing a synthesizable, lintable, and readable end product.

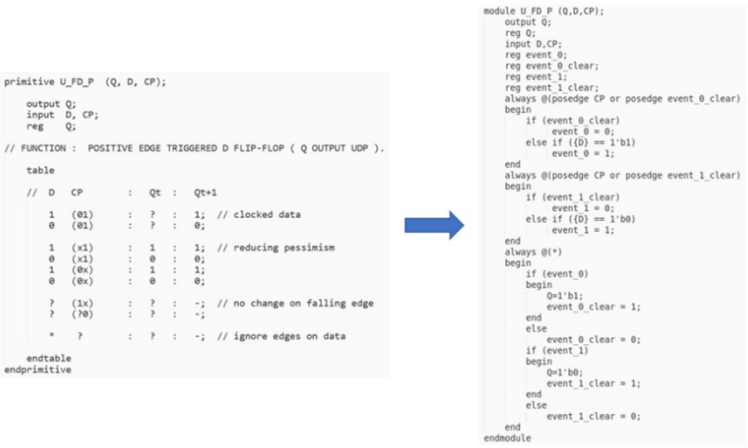
Once a suitable target has been produced, a number of parsing steps as indicated in **Figure 1** are required before an analysis can be done. The initial parsing stage for netlists is considerably less difficult than in software. Since machine code and data are stored together in the same binary format, an outstanding problem in software is the impossibility of realizing a parser that can distinguish code from data, particularly for architectures with arbitrary instruction lengths such as Intel x86. Netlists, unlike machine code, are typically represented in a hardware description language (HDL) and can be easily parsed with a standard HDL parser to produce an AST. Verilator [4] was chosen to conduct this process since it is open source, robust, and produces an XML AST while requiring only a small amount of preprocessing of the netlist.

In both software and hardware, the semantic representation used to produce an AST should be as language independent as possible. This is necessary in software decompilation as assembly languages have idiosyncrasies such as branch delay slots and seemingly simple instructions such as call (i.e. jump) which change several registers simultaneously. In fact, it is common practice for decompilers to have their own intermediate representation language to further improve syntax analysis in the conversion from assembly to source code [3]. Netlists suffer from similar issues in that they contain standard cells or LUTs and, if recovered directly from a bitstream or a synthesis byproduct, IP blocks. A further obstacle are primitives which are defined by directly specifying state transitions; therefore, they must be abstracted before the analysis can occur. Since most of a Verilog netlist consists of basic logic, the choice was made to simply reduce complex statements to simpler Verilog rather than create an IL.

Two preprocessing steps are required before using Verilator. First, primitive tables are converted to behavioral code using a heavily modified version of PyVerilog. This introduces events and both improves readability and simplifies extraction of data and control flow during the analysis phase. Second, the netlist hierarchy is flattened by replacing all submodule instances with their respective code, replacing IO with the corresponding wires from the top-level module to reduce the number of unnecessary variables. Flattening the netlist in this way allows for easier matching across equivalent designs.

When the source netlist is initially read into the parsing script, a list of all sub-modules instantiated within the design is generated. The script then looks for the definitions of these sub-modules within either the source netlist file, or a helper library file also provided as an argument to the script. All sub-module definitions are ultimately appended to a preprocessed version of the source netlist used in later stages of the parsing workflow.

It may be the case that the top-level module contains instances of what are known as Verilog user-defined primitives rather than, or in addition to, modules. While a user-defined primitive and a module are similar in that both can be viewed as sub-structures that may be instantiated as part of a top-level design, they differ in how their behavior is implemented as Verilog code. User-defined primitive definitions are coded using Verilog 'table' constructs. The table code separates the primitive's inputs and outputs on the left and right. Each row of the table specifies some combination of input values, or edge transitions. The right side of the table specifies the value or transition of the output corresponding to those input conditions. To better illustrate this concept, the simple user-defined primitive definition of a D Flip Flop is shown in **Figure 2**. To get a clear example of how the function of the primitive is defined in terms of the rows of the table, we can look at the first row of the table. Under the 'D' column we see '1' and under 'CP' (the clock signal), we see '(01).' This row therefore specifies how the output of 'Q' transitions under the condition that the value of 'D' is 1 and the clock is making a positive edge transition. By looking at the right side of the table, we find '?' under the 'Qt' column and '1' under the 'Qt+1' column. The entire row can thus be interpreted as specifying that when 'D=1' and the clock makes a positive edge transition, the next value of the output 'Q' will be 1 ('Qt+1 = 1') regardless of its previous value ('Qt = ?'). This is exactly the behavior we would expect of positive edge triggered D Flip Flop.



```
primitive U_FD_P  (Q, D, CP);

    output Q;
    input  D, CP;
    reg    Q;

// FUNCTION :  POSITIVE EDGE TRIGGERED D FLIP-FLOP ( Q OUTPUT UDP ).

    table

    //  D   CP       :   Qt  :   Qt+1

        1   (01)     :   ?   :   1;  // clocked data
        0   (01)     :   ?   :   0;

        1   (x1)     :   1   :   1;  // reducing pessimism
        0   (x1)     :   0   :   0;
        1   (0x)     :   1   :   1;
        0   (0x)     :   0   :   0;

        ?   (1x)     :   ?   :   -;  // no change on falling edge
        ?   (?0)     :   ?   :   -;

        *   ?        :   ?   :   -;  // ignore edges on data

    endtable
endprimitive
```

```
module U_FD_P (Q,D,CP);
    output Q;
    reg Q;
    input D,CP;
    reg event_0;
    reg event_0_clear;
    reg event_1;
    reg event_1_clear;
    always @(posedge CP or posedge event_0_clear)
    begin
        if (event_0_clear)
            event_0 = 0;
        else if ({D} == 1'b1)
            event_0 = 1;
    end
    always @(posedge CP or posedge event_1_clear)
    begin
        if (event_1_clear)
            event_1 = 0;
        else if ({D} == 1'b0)
            event_1 = 1;
    end
    always @(*)
    begin
        if (event_0)
        begin
            Q=1'b1;
            event_0_clear = 1;
        end
        else
            event_0_clear = 0;
        if (event_1)
        begin
            Q=1'b0;
            event_1_clear = 1;
        end
        else
            event_1_clear = 0;
    end
endmodule
```

Figure 2. D Flip-Flop Primitive Converted to Verilog Module with Procedural Blocks

A key phase of the preprocessing step involves converting any user-defined primitives into equivalent module definitions. This not only regularizes the structure of the XML Syntax Tree, but also defines the behavior of the primitive making use of

procedural **always** blocks rather than tables. The use of procedural blocks allows for an easier construction of the sequential graph of the device later in the workflow. **Figure 3** displays a user-defined primitive instance of a purely combinational 2-to-1 multiplexer that has been converted into an equivalent module definition. Note that in the case of a purely combinational primitive design, no procedural blocks are necessary within the module definition as the device has no edge sensitivities. Instead, the behavior of the device can be described using a single continuous **assign** statement for each output. Once all sub-modules and user-defined primitives have been converted and appended to the preprocessed file, a call to Verilator is made on the preprocessed file to produce the syntax tree of the Verilog code in XML format. Once the syntax tree has been created, it is converted into a Python Ordered Dictionary structure and passed to the flattening script.

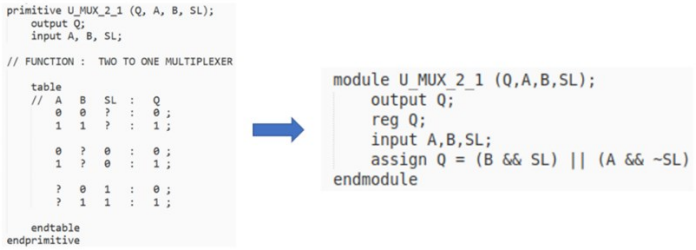


Figure 3. 2-to-1 Multiplexer Converted to Module with Single Continuous Assign Statement

The role of the flattening script is to replace any sub-module instances within the top-level module with their respective behavioral code. To do this, the script performs a depth-first walk of the syntax tree dictionary. When a sub-module instance is found within a higher-level module, one of two things may occur. If the submodule instance has itself already been parsed, the contents of the sub-module definition are copied into the higher-level module's syntax tree and all variable names within the module definition are replaced with the corresponding port names in the higher-level module. The sub-module instance is then removed from the syntax tree of the higher-level module. If the sub-module has not yet been parsed, a recursive call is made to the parsing function on the sub-module instance. In this way, the top-level module is recursively flattened from the bottom up.

Two directed graphs are generated from the AST. The first consists of sequential or blocking operations and corresponds roughly to the control and data flow graphs for software. The second is the combinational logic, corresponding to basic blocks in software. Sequential graphs are useful for bus identification and hierarchy reconstruction. Combinational graphs provide a structure for signature comparison. Boolean satisfiability (SAT) solver-based techniques for signature matching have been known since the 1980s. Further, apart from asynchronous designs, all combinational graphs are trees. Tree isomorphisms are efficient, unlike SAT solvers or loopy graph isomorphisms. Trees also lend themselves to fuzzy matching algorithms not available to general graphs.

The sequential graph of the design is formed by adding a node for each procedural **always** block found within the flattened Verilog file. Any nets appearing in the edge condition, within if conditionals, or appearing on the right side of **assign** statements within the **always** block are considered inputs to the block. A node is added to the graph for each input pointing *from* the input node *to* the **always** block node. Alternatively, any nets assigned a value within the **always** block (those appearing on the left side of assign statements) are considered outputs to the block. A node is added to the graph for each output pointing *from* the **always** block node, *to* the output node. This method of constructions is illustrated in **Figure 4**.

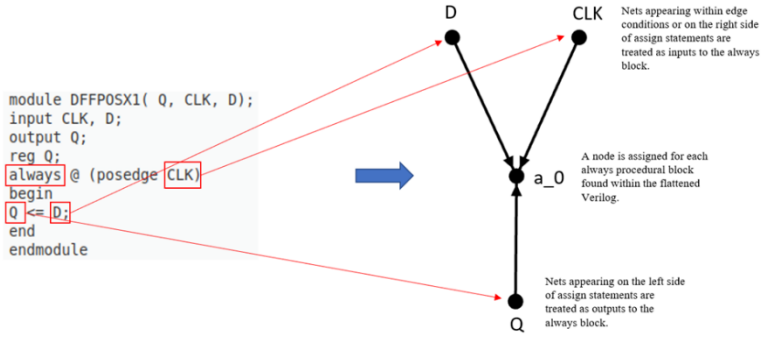


Figure 4. Procedural `Always` Block Graph Representation

A typical netlist will incorporate possibly thousands of procedural **always** blocks. A good method of isolating the registers of the design is to remove the clock signal from the generated sequential graph and separate all connected components. The resulting 'islands' seen in the graph correspond to the different registers in the design. This is illustrated in the **Figure 5**. Removing the clock signal can be done either manually or heuristically. For most sequential graphs clock, reset, and other global synchronizing events are very heavily utilized. This flows from the definition of a sequential graph. A threshold can thus be set for degree of nodes and all nodes above that degree can be removed. In practice the gap is so large, a six-sigma threshold is not unreasonable by default, but this can be arbitrarily adjusted.

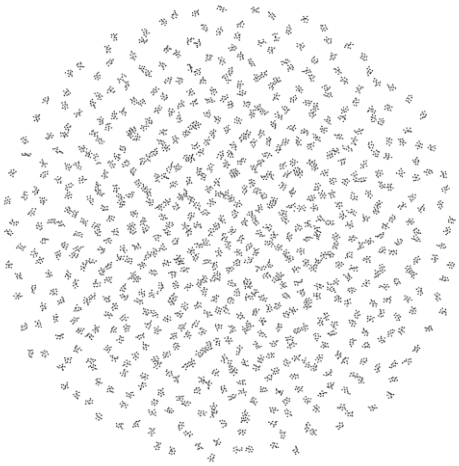


Figure 5. Connected Components of the Sequential Graph Representing Individual Registers

Generally, the fan-out of a given connected component representing an **`always`** block will feed into the combinational logic of the design as an input. Alternatively, the inputs to the **`always`** block feed out of the combinational logic as outputs, and thus a loop is formed between the combinational logic and memory portion of the design. Furthermore, considering the combinational 'path' between the fan-in and fan-out of a given **`always`** block provides a potential means of register mapping between similar designs. The combinational graph of the design represents all combinational logic in the form of a directed acyclic tree. The graph is formed by assigning nodes to all nets, wires, and operators appearing in continuous **`assign`** statements within the flattened Verilog file. An edge is drawn *from* any net or wire *to* the operator which acts upon it. Similarly, an edge is drawn *from* any operator *to* any wire which has its value assigned the result of the operator acting on other nets in the design. The combinational graph constructed is illustrated in the **Figure 6** of an OAI22X1 cell which is comprised of two ORs, one AND, and one NOT operation.

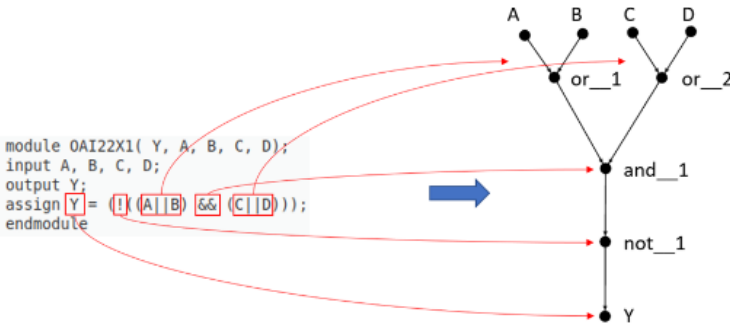


Figure 6. Combinational Graph Representation of OAI22X1

While signature matching of logical structures may be easier in hardware than in software, registers are one of the most difficult structures to recover in hardware. Register recovery is akin to array/struct recovery in software, but unlike an array in software, which can be inferred by specific events such as a **`malloc()`** or **`memcmp()`** subroutine call, registers must be found by analyzing the netlist graph structure. This work leverages the RELIC algorithm [5][15] as one method to detect registers. RELIC assumes a group of flip-flops with similar fan-in structures are likely to be in the same word of a register, whereas a group of flip-flops with vastly different fan-in structures are unlikely to be related. Nodes going out of or into flip-flops always form roots or leaves respectively in the combinational graph. Finding fan-in and fan-out then consists of finding all paths through the tree which connect to these nodes. The RELIC algorithm approximates graph isomorphism by computing a fuzzy similarity score between logic structures. By comparing the similarity of the fan-in structure of every pair of flip-flops in a netlist, an undirected graph can be created to visualize the relationships between individual registers. In this similarity graph, flip-flops in the original netlist are represented as nodes, with edges between them indicating a similarity score greater than zero and the edge weight relating to the similarity score. When low-weighted edges are filtered out, the resulting graph contains separate connected components, with each connected component indicating a group of similar flip-flops. Using this method, RELIC was able to successfully identify all bits of multi-bit registers in many test cases but returns false positives or fails to recognize relationships in others.

A more robust register detection method would aggregate results from multiple algorithms in addition to RELIC to accurately recover register bit groupings across situations where a single algorithm would fail to yield useful results. To this end, other methods of identifying multi-bit dataflow structures are also being explored, such as identifying counter or multiplexer structures. Identified word-level structures such as these reveal information about busses, which can be propagated forward and backward through the netlist to identify other word-level structures connected to the identified components. Identifying busses and word-level objects in a netlist is a crucial step for converting netlists back into behavioral RTL. It elevates the conversion process to a higher level of abstraction, with a focus on identifying word-level operations and data path recovery rather than identifying single-bit structures floating in a sea of gates. This makes components of the design more human-identifiable and is a necessary step for recognition of higher-level modules and behavior.

Software is delimited by basic blocks. A basic block is a section of code wherein all instructions are executed sequentially without repetition or interruption. These can be identified by looking for instructions which interrupt control flow. For example, by identifying the setup instructions of a function, you can identify the beginning of a basic block, as it would be the target of a call (i.e. jump) instruction. Similarly, the instructions which terminate a function typically signify the end of a basic block since the function must return control to another basic block. Ultimately, identifying these parts allow for one to identify the basic blocks which are being used to generate a control-flow graph of the program. The software decompiler then just needs to use its knowledge of the program structure and the syntax to generate the higher-level source code.

B. Combinational and Sequential Graph Analyses

Hierarchy and module reconstruction in hardware are a difficult and subjective problem. Module reconstruction does not have a good analog in software since hardware lacks the delineating instructions. To gain some traction, hierarchy reconstruction in hardware is addressed by computing maximal directed cycles in the sequential graph, as is done for reintroducing loops in software. Structures with I/O only consisting of identified registers or control bits (such as clock or reset), or that appear repeatedly, are labeled as modules. This still has limited success for two reasons. First, instances of modules are not repeated with near the frequency of function calls in software. Secondly, mining subgraphs is known to be a nondeterministic polynomial time (i.e. NP-complete) problem. Fortunately, both the software and hardware community reuse code heavily. Signature matching or finding smaller known functions within an unknown code base is computationally

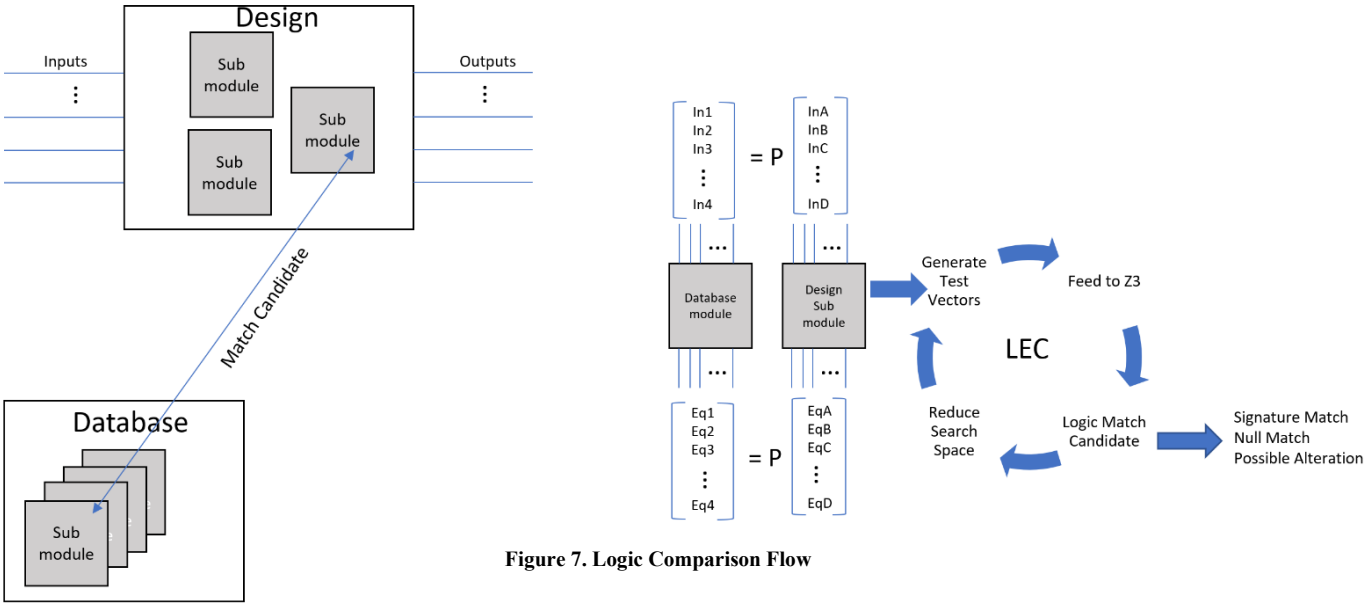


Figure 7. Logic Comparison Flow

much more feasible than directly mining the entire graph for the unknown codebase.

The application BinDiff [10] is one of many software tools which provide this functionality. According to [11], BinDiff works on the abstract structure of an executable, ignoring the concrete assembly-level instructions in the disassembly. The structure of the functions and basic blocks (i.e. the features of the control flow graph) are what is used for analysis. For example, BinDiff can count the call graph edges, or the number of calls to get to a function in a binary and compare it with functions in the other binary to see if any match. By repeating this procedure on all the functions in the two files, BinDiff gets a score for how similar the two binaries are overall. Similarly, netlists are generally a composite of modules with specific functionality such as adders, timers, and multiplexers. By employing the same techniques that BinDiff uses, the modules making up a netlist can be compared with a large set of pre-analyzed netlists of common hardware components in order to find a match. The applicability of these algorithms for this task works well since BinDiff was designed to work on different architectures and across different compilers with different optimizations. As such, netlist comparisons must be able to work with respect to different platforms, implementations, and manufacturing intricacies. Potential module matches identified through subgraph matching can be confirmed using a SAT solver. The AST of the design is walked through to produce a set of Boolean logic equations that define the combinational logic for the design. These equations are formatted for the Python Z3 API and stored in a file. Z3 files for database entries are stored and are later used to assist in signature matching. The file contains logic that defines the combinational relationship between inputs and outputs. It will also include internal wires defined in the flattened Verilog file. **Figure 7** presents a flow diagram that illustrates the full logic flow comparison. In instances where these wires are not needed, the equation set can be reduced using a composition of the corresponding Boolean logic equations. Control functions of the logic equation data structure provide a means to perform operations on the structure. A list of a possible links between inputs and outputs in the two graphs can be generated. These sets are then used to generate code to run those pairing assertions in Z3.

Components and modules that comprise a larger design can be identified through signature matching. In this case, combinational subgraphs of interest from the larger design can be used to produce a bounding set on the search space. This aligns itself well with the overall task of extracting RTL components while also helping limit scalability issues in designs with many nodes. Once a subgraph has been identified, its Z3 file is used to populate the logic data structure with the corresponding equations. From there, the Z3 files from the database are read into a separate logic data structure. Data structure utilities also allow for the generation of a set of all partially composed equation from the original set to be produced that match the input and output structure of the signature match candidate. This allows for some flexibility if the subgraph has additional or missing nodes that might cause a failure to match.

C. *Module Recovery and Generation of RTL*

Modules are loaded into a LEC script that populates a data structure with the logic equations. The equations are stored and referenced by a key value corresponding to their node in the graph. The terms in the equation are replaced with dummy variables and a list is created to maintain a link between the dummy variables and the term names in the design. **Figure 8** shows the extracted logic and how it is represented in the logic database.

```
count[0x0] == (T1)
_0___1 == ( T1 ^ 1 )
_0___2 == ( count[0x1] ^ T1 )
_0___3 == ( count[0x2] ^ ( T1 & count[0x1] ) )
_0___4 == ( count[0x3] ^ ( count[0x2] & ( T1 & count[0x1] ) ) )
T0 == (1)

{count[0x0]: ['({})', ['T1']]}
{_0___1: ['( {} ^ 1 )', ['T1']]}
{_0___2: ['( {} ^ {} )', ['count[0x1]', 'T1']]}
{_0___3: ['( {} ^ ( {} & {} ) )', ['count[0x2]', 'T1', 'count[0x1]']]}
{_0___4: ['( {} ^ ( {} & ( {} & {} ) ) )', ['count[0x3]', 'count[0x2]', 'T1', 'count[0x1]']]}
{T0: ['(1)', []]}
```

Figure 8. The extracted logic (*top*) and the structural representation in the logic database (*bottom*)

The first pass will attempt to identify any equations defining an input-output relationship with the same structure between the design and the database module. The equations in the data structures are filled with dummy variables and compared against the equations from the database module. The equations are compared by a generated set of Z3 assertions and executing Z3 files during run time. In this first past, each test is done individually without knowledge of previous test assertions or matches. If a match is found, the equations are linked in a list, the relationship of input wires between the database graph and subgraph are stored, and the corresponding node names for the terms are linked. The process will continue and any further matches will also be added to the list. At the end of the first pass, the control function will be handed back a list of possible match candidates. Many of these matches will contain large numbers of candidates due to the common occurrence of equations relating to single gate structures. In the next phase, the control code will identify any one-to-one match, link those nodes from the design to the database module, and further limit the search space. The control code will move progressively through the list from small matching sets to larger sets. At each step, a system of the matching sets will be used to try to further reduce the search space. If a pairing between inputs produces a match for the logic of two outputs, that pairing is then used as a candidate for matching other outputs. Input pairings are scored based on the number of output matches they are able to produce. An exact match for all outputs, given an input pairing, would yield a signature match for the module and the subgraph could be replaced with the database module. Partial matches are also useful in pointing to the need to redefine the subgraph or spotting modules in the design that contain small alterations that should be checked.

IV. Experimental Results

A. Data Acquisition and Benchmarking

Several test articles were developed to use in experiments for collecting empirical data. Python scripts for generating the sequential graphs were run on a Serial Peripheral Interface (SPI) netlist that was extracted from a fabricated layout [16]. The SPI consisted of roughly 400 cells, including 63 flip flops. The graph was generated in 9.13 seconds on an Ubuntu 18.03 virtual machine run on a host computer with an Intel Core i5-835U CPU running at 1.7 GHz. The graph contained 806 nodes and edges and revealed the flip flop structures of the extracted netlist. Signature matching was done with the Python Z3 API for exact matches and Levenshtein Distance for fuzzy matching due to convenience of implementation [6].

This framework also makes comparative validation easier. To this end, sequential and combinational graphs were constructed for a floating-point unit (FPU) golden test article along with the same FPU design with inserted malicious circuitry. The inserted circuitry was designed to break the multiplication design pipeline after a specific clock cycle count. Differences between the two graphs were extreme as shown in **Figure 9**. The graphs are analyzed based on *degree* (i.e. the number of nodes/edges connected to the node being analyzed.) The original design on the left is dominated by the clock signal which has degree 1384 in the sequential graph and the inverted reset in the combinational graph with degree 537. All flip-flops also are identical and contain 7 nodes. However, the design containing the malicious insertion on the right has a combinational graph with 13 nodes of degree between 46 and 86 including reset and a sequential graph with 6 nodes of degree between 170 and 174.

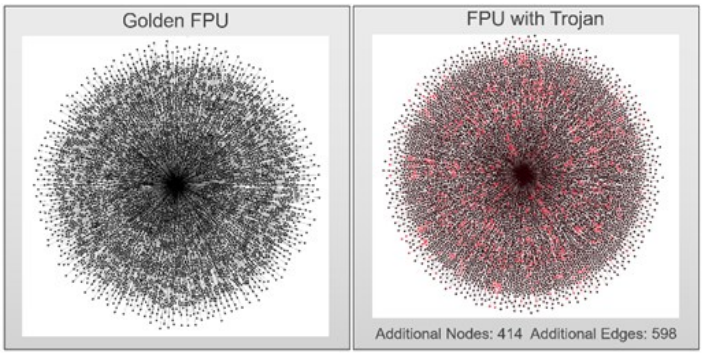


Figure 9. Sequential graphs of the golden FPU (*left*) and FPU with malicious insertion (*right*). The additional nodes are red.

One notable observation was the single flip-flop with 8 nodes, the extra being the reset. Upon deeper inspection, this is a differential flip-flop which asserts reset on an event. While the event is not trivial to recover with the current maturity of the toolset, this behavior is indicative of a denial-of-service-like Trojan that would reset the FPU.

In order to obtain metrics for real world devices with different sized netlists, the workflow was completed for a SPI module, a floating-point unit, floating-point processor, and AES core. The results of these runs are displayed in Table I. The metrics help to assess the scalability of workflow over larger netlists as a design grows in complexity. The tests were run on the same Ubuntu 18.03 virtual machine from the host computer with an Intel Core i5-835U CPU running at 1.7 GHz.

TABLE I. PARSING SCRIPT PERFORMANCE METRICS

	TEST ARTICLE DESIGN BENCHMARKS			
	SPI Interface (small)	**FPU (medium)**	**Processor (large)**	**AES Core (very large)**
Preprocessing Time	0.0692s	1.3585s	3.248s	10.1s
Flattening Time	0.6463s	27.7335s	304.4473s	930.345s
Graph Generation Time	0.3065s	2.0682s	16.1446s	98.56s
Z3 Equation Generation Time	0.0067s	2.0969s	9.2735s	15.645s
Total Execution Time	**1.1276s**	**34.1421s**	**334.0910s**	**1054.65s**
Combinational Nodes	1,156	35,819	84,429	235,678
Combinational Edges	1,316	47,634	111,434	345,543
Sequential Nodes	2,420	6,229	27,519	67,786
Sequential Edges	4,278	8,996	39,663	89,864

B. *Mapping Non-database Modules to Known Database Structures*

An 8-bit counter was compared to a 4-bit toggle flip-flop counter within the database. The Verilog files for the designs are shown in **Figure 10**. Once the 8-bit counter was put through the workflow, the Z3 files were created and compared to the database entry Z3 file of the 4-bit counter. A matching subgraph was determined for the lower 4 bits of the 8-bit counter. From there, the modules were loaded into the LEC script. **Figure 11** presents the database counter and subgraph after having populated the logic data structure. From here the control code will search for logically equivalent equation structures using dummy variables. The control code will pass parameters to the code generator to develop Z3 tests. If the tests show that the equations are never different, a match is made. One iteration of this is shown below. Once the process is complete, a list of matches is returned along with an output of any one-to-one matched equations.

```
module top(clk, reset, count);
input clk, reset;
output[3:0] count;

//reg [3:0] count;
wire clk, reset;

wire T3;
wire T2;
wire T1;
wire T0;

assign T3 = count[0] & count[1] & count[2];
assign T2 = count[0] & count[1];
assign T1 = count[0];
assign T0 = 1'b1;

TFF_async_clrN reg3(.T(T3),.clk(clk),.clrN(reset),.q(count[3]));
TFF_async_clrN reg2(.T(T2),.clk(clk),.clrN(reset),.q(count[2]));
TFF_async_clrN reg1(.T(T1),.clk(clk),.clrN(reset),.q(count[1]));
TFF_async_clrN reg0(.T(T0),.clk(clk),.clrN(reset),.q(count[0]));


endmodule
```

```
module top(clk, reset, count);
input clk, reset;
output[7:0] count;

reg[7:0] count;
wire clk, reset;

always@(posedge clk or negedge reset)
begin
if (~reset)
    count = 8'b0;
else
    count = count + 8'b1;

end
endmodule
```

Figure 10. 4-bit counter module (*top*) from database and the 8-bit design match target (*bottom*).

```
{_00_[0x0]: ['(~ {})', ['count[0x0]']]}
{_00_[0x1]: ['( {} ^ {} )', ['count[0x1]', 'count[0x0]']]}
{_00_[0x2]: ['(~( ( {} ^ (~( ( {} & {} )) )))', ['count[0x2]', 'count[0x1]', 'count[0x0]']]}
{_00_[0x3]: ['(~( ( {} ^ (~( ( {} & {} )& {} )) )))', ['count[0x3]', 'count[0x1]', 'count[0x0]', 'count[0x2]']]}

{_0___1: ['( {} ^ 1 )', ['T1']]}
{_0___2: ['( {} ^ {} )', ['count[0x1]', 'T1']]}
{_0___3: ['( {} ^ ( {} & {} ) )', ['count[0x2]', 'T1', 'count[0x1]']]}
{_0___4: ['( {} ^ ( {} & ( {} & {} ) ) )', ['count[0x3]', 'count[0x2]', 'T1', 'count[0x1]']]}
{T0: ['(1)', []]}
```

Figure 11. Subgraph output logic (*top*) and database module logic(*bottom*) as stored in logic data structure.

```
y1 = BitVec('y1',1)
y2 = BitVec('y2',1)
var4 = BitVec('var4',1)
var3 = BitVec('var3',1)
var2 = BitVec('var2',1)
var1 = BitVec('var1',1)
s.add(y1 == ( var4 ^ ( var3 & ( var2 & var1 ) ) ))
s.add(y2 == (~( ( var4 ^ (~( ( var3 & var2 )& var1 )) ))))
s.add(y1 != y2)
goutput = s.check()
```

```
_0___1 == ( T1 ^ 1 )
_00_[0x0] == (~ count[0x0])


_0___2 == ( count[0x1] ^ T1 )
_00_[0x1] == ( count[0x1] ^ count[0x0] )


_0___3 == ( count[0x2] ^ ( T1 & count[0x1] ) )
_00_[0x2] == (~( ( count[0x2] ^ (~( ( count[0x1] & count[0x0] ))) )))


_0___4 == ( count[0x3] ^ ( count[0x2] & ( T1 & count[0x1] ) ) )
_00_[0x3] == (~( ( count[0x3] ^ (~( ( count[0x1] & count[0x0] )& count[0x2] )) )))
```

Figure 12. Example of a Z3 assertion set (*top*). After all test vectors have completed the set of match candidates is returned (*bottom*).

V. Conclusion

In this paper, a workflow for decompilation of a gate-level netlist into a human readable RTL Verilog file was presented. Software decompilation methodologies were leveraged and enabled this process although techniques and the challenges vary between software and hardware. The well-established software decompilation process is ripe with further techniques which can be added to this workflow in future research.

Acknowledgment

All of the research presented in this paper was funded by Wright-Patterson Air Force Research Lab in Dayton, OH.

References

[1] Quijada, R., Dura, R., Pallares, J. et al. J Hardw Syst Secur (2018) 2: 322. https://doi.org/10.1007/s41635-018-0051-4

[2] Kimura, A., Scholl, J., Schaffranek, J., Sutter, M., Elliott, A., Strizich, M., Via, G., Journal of Hardware and Systems Security (2019)

[3] C. Cifuentes, “Reverse Compilation Techniques,” Phd diss. Queensland University of Technology, 1994.

[4] Verilator. (4.000, 2018), Wilson Snyder. Available: https://www.veripool.org/wiki/verilator

[5] T. Meade, J. Yier, M. Tehranipoor, S. Zhang. "Gate-level Netlist Reverse Engineering for Hardware Security: Control Logic Register Identification", ISCAS, pp. 1334-1337, 2016.

[6] Z3. (4.8.7), N. Bjorner, L. de Moura, L. Nachmanson, C. Wintersteiger. Microsoft Research. Available: https://github.com/Z3Prover/z3/

[7] Ilfak Guilfanov, “Decompilers and beyond.” Hex-Rays .com. Hex-Rays SA, 2008. 10 Oct. 2019 https://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf

[8] https://www.program-transformation.org/Transform/DecompilationProcess

[9] Ilfak Guilfanov. “Decompiler internals: microcode.” RECON 2018. Brussels, Belgium. http://www.hex-rays.com/products/ida/support/ppt/recon2018.ppt.

[10] zynamics. “BinDiff.” www.zynamics.com/bindiff.

[11] zynamics. “BinDiff Manual.” www.zynamics.com/bindiff/manual/index.html#chapUnderstanding.

[12] J. Koret. “Diaphora.” https://diaphora.re.

[13] Benz, F., Seffrin, A., Huss S., *BIL:A tool-Chain for Bitstream Reverse-Engineering*, IEEE, 978-1-4673-2256-0/12, 2012.

[14] Geraf, J., Harper, S., Lerner, L., Ensuring Design Integrity through Analysis of FPGA Bitstreams and IP Cores, International Conference on Reconfigurable Systems and Algorithms, 2012.

[15] Brunner, M., Baehr, J., Sigl, G., *Improving on State Register Identification in Sequential Hardware Reverse Engineering*, IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 978-1-5386-8064-3/19, 2019.

[16] A. Kimura, A. Waite, J. Scholl, J. Schaffranek, G. D. Via, "From Silicon to Simulation: A Full Decomposition of a Fabricated 130 nm Serial Peripheral Interface for Establishing an Assurance Baseline Root-of-Trust", *Physical Assurance and Inspection of Electronics (PAINE)*, December 2020.

[17] A. Kimura, A. Waite, J. Scholl, G. Via, "Applied Failure Analysis Tools and Techniques Towards Integrated Circuit Trust and Assurance", ASM International, *Electronic Device Failure Analysis*, Volume 23 No.1, 2020.