Bitstream Assurance Checking Engine for Undocumented Functionality (BRACE)

Andrew G. Schmidt, Justin Wilford, Benedict Reynwar, Ting-Yuan Sung, and Matthew French
Information Sciences Institute
University of Southern California
Arlington, VA, USA
{aschmidt, jwilford, breynwar, tsung, mfrench}@isi.edu

***Abstract*— Assurance that a modern FPGA is secure is difficult to achieve as the manufacturers do not disclose all user-accessible circuitry that physically exists on the device. This "undocumented functionality" is invisible to the end-user and makes it extremely difficult to perform a security analysis of the device. These undocumented features are typically circuitry that a vendor does not wish to support or disclose to end users; however, they can be vulnerabilities that a bad actor in our global marketplace could exploit. To mitigate these undocumented features, the University of Southern California, (USC/ISI), is developing BRACE, the *B*itst*R*eam *A*ssurance *C*hecking *E*ngine for undocumented functionality. BRACE is an automated tool to detect user reachable undocumented states in COTS FPGA hardware and assures bitstreams are prevented from using potentially malicious undocumented functionality. BRACE creates a database of known and unknown settings and their mapping to a bitstream, which can then be utilized for both static bitstream checking during bitstream design and device programming, and runtime Hard IP control checking during device operation. These checks are lightweight, operate within seconds to minutes, and do not require a human analyst in the loop. BRACE is an open framework that can be extended to host other 3rd-party bitstream assurance analyses, such as automated wiring short detectors or other bitstream security tools. By providing both static and runtime protection mechanisms, BRACE protects COTS FPGA bitstreams through their entire life cycles.**

I. Introduction

The objective of the **B**itst**R**eam **A**ssurance **C**hecking **E**ngine for Undocumented Functionality **(BRACE)** is to provide hardware assurance against unknown features within the Very Large Scale Integration (VLSI) of Field Programmable Gate Arrays (FPGAs). FPGAs are used pervasively throughout DoD systems, representing 33% of DoD microelectronics expenditures [1]. FPGAs are complex devices, whose creation is overseen by fabless semi-conductor companies who use a global ecosystem to create, manufacture, and supply them. It is common for FPGA vendors to include functionality that is not disclosed to the end user for self-test, advanced proto-typing, cost savings, and to hide errata. These undocumented features can be exploited as hardware trojans during either design or runtime.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited (PA# AFRL-2021-3136)

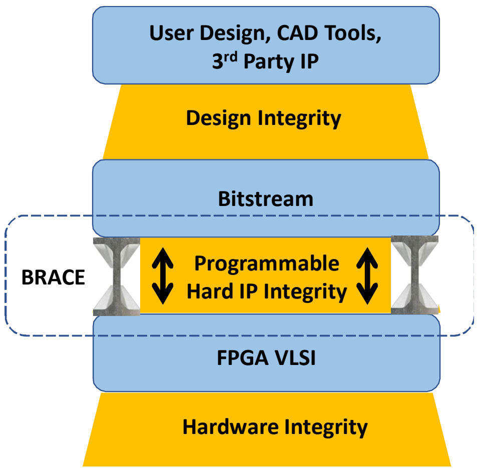

User Design, CAD Tools, 3rd Party IP
Design Integrity
Bitstream
BRACE
Programmable Hard IP Integrity
FPGA VLSI
Hardware Integrity

Figure 1- FPGA Integrity Elements

FPGAs are critical to the DoD as they are used pervasively throughout defense systems for their ability to deliver near custom-ASIC level performance at much lower cost, while their programmability enables future capability upgrades and errata fixes. These benefits come with the tradeoff that these are complex devices that are developed by commercial companies that may not make their designs fully available for outside analysis, making FPGA assurance a difficult challenge. Figure 1 depicts the three principal components that comprise the FPGA ecosystem, their relative relationship to one another, and the layers of assurance that are needed to provide a comprehensive FPGA assurance solution. The first aspect is the user's design, 3rd party IP, and the CAD tools that the user collectively uses to create their application circuit and map it into an FPGA programming file, called a bitstream. Prior art, such as the FPGA Design Integrity tools developed by MacAulay Brown under the DARPA Trust and Vet efforts, has developed advanced techniques to ensure the bitstream is free of any hardware trojans residing in the user's circuit level representation in the bitstream itself. The bottom aspect is the FPGA VLSI device itself, which for assurance purposes can be viewed as an ASIC. Here many of the hardware imaging techniques developed under the DARPA Trust and IRIS programs can be utilized.

FPGAs however, introduce a third, middle layer that is not often seen in ASICs, which is the interaction, especially during run time, between the bitstream, control settings, and the underlying FPGA VLSI. As FPGAs have matured, they have added increasingly complex Hard IP to the device to incorporate functions that end users are commonly utilizing, to save device area and increase performance. Hard IP has grown in complexity from simple multiple accumulators to full DSP units, and now include a range of functions from complex I/O standards, tri-mode ethernet media access controllers (TEMAC), internal configuration access port (ICAP), digital clock managers, and system monitors. Each of these Hard IP primitives have control settings that are set either in the bitstream during configuration or during runtime through user state machines and control signals. In practice, many of these control settings are not fully documented. These undocumented features may enable built in self-test features only used for acceptance testing, enable advanced features that a user needs to pay extra for, be proto-types for next generation devices, or be a deprecated feature due to an erratum. Due to the extreme sensitivity to VLSI cost and area overhead, FPGA vendors predominately perform safety and security checking in software-based CAD tools using design rule checks (DRCs) rather than enforcing configuration rules directly within the VLSI. Furthermore, vendors rely on user guides to inform FPGA application designers how to control Hard IP and leave the responsibility of ensuring correct operation to the designer.

II. Technical approach

BRACE is developing an automated toolset, illustrated in Figure 2, to identify undocumented modes and prevent their activation during design time, as well as runtime after deployment. The key technology of BRACE is a database of FPGA Hard IP settings and their mappings to FPGA bitstreams [2]. The database is developed utilizing USC/ISI's Tools for Open Reconfigurable Computing (TORC) [3] and leverages open-source FPGA bitstream documentation tools [4,5] to trace the bitstream mappings. Static bitstream checkers then utilize the database to ensure an FPGA bitstream is free of undocumented functionality by iterating over the device and validating the configuration settings of FPGA Hard IP. Integrity Assurance Wrappers similarly leverage the database as runtime checkers capable of preventing or alarming if Hard IP control is set to illegal values during runtime. The BRACE approach addresses assurance for 100% of the user accessible Hard IP types on an FPGA device. The BRACE framework currently is in development to support Xilinx FPGAs, but the concept is inherently adaptable to other Xilinx and Intel (Altera) devices.

BRACE seeks to deliver a robust prototype tool that can be rapidly transitioned into operational systems upon completion of the project. To enable successful transition and to increase trust in the final product, BRACE utilizes a robust set of validation and testing efforts. The primary goals of testing and validation efforts are to 1) demonstrate the completeness and correctness of the BRACE database 2) demonstrate the accuracy and runtime performance of the static checker, 3) demonstrate the correctness and sensitivity of the Integrity Assurance Wrappers and 4) demonstrate that the software frameworks perform as expected and are reasonably resilient.

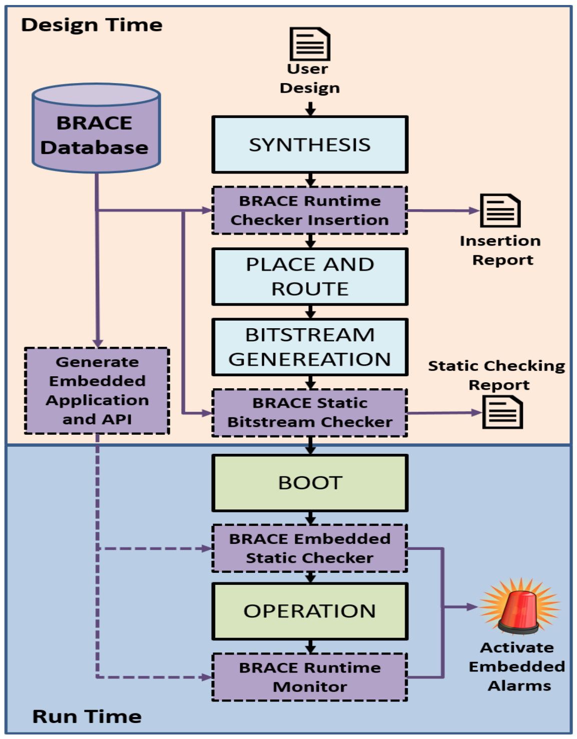
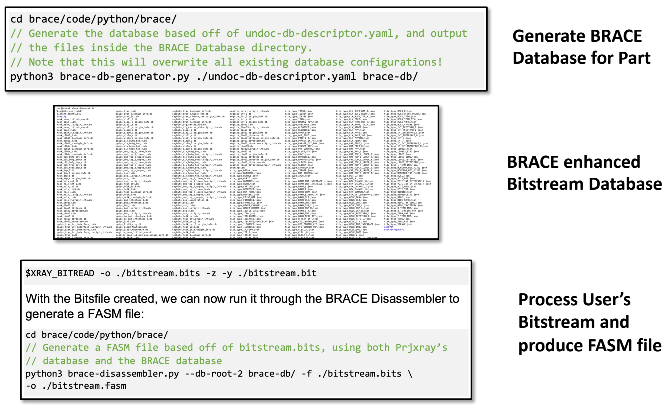


Figure 2 - High-Level BRACE design and operation flow

The completeness of the BRACE database will be validated by calculating the *Database Completeness* metric. Once this metric has been computed, the correctness of the database will be verified by generating *all* documented configuration modes and ensuring that these modes are not contained in the BRACE database (only undocumented modes should be present). Then *all* undocumented modes will be generated and tested to ensure that these modes are present in the database. Exhaustive testing is feasible given the large degree of parallelism in the bitstream structure.

A. *BRACE Static Bitstream Checking*

Once a bitstream is generated there no longer is a vendor provided mechanism to validate the configuration settings against the original user's design. With BRACE's Static Bitstream Checker a comprehensive tool suite is provided to read, analyze, and report on the bitstream configuration for Hard IP. Moreover, BRACE's framework provides the ability to report all Hard IP the configuration settings and their relevant values in the User's Guide. For assurance purposes, BRACE goes beyond the known and documented configuration settings and reports out the unknown, undocumented, or illegal states.


Generate BRACE Database for Part
BRACE enhanced Bitstream Database
Process User's Bitstream and produce FASM file

```
cd brace/code/python/brace/
// Generate the database based off of undoc-db-descriptor.yaml, and output
// the files inside the BRACE Database directory.
// Note that this will overwrite all existing database configurations!
python3 brace-db-generator.py ./undoc-db-descriptor.yaml brace-db/
```

```
$XRAY_BITREAD -o ./bitstream.bits -z -y ./bitstream.bit
```

With the Bitsfile created, we can now run it through the BRACE Disassembler to generate a FASM file:

```
cd brace/code/python/brace/
// Generate a FASM file based off of bitstream.bits, using both Prjxray's
// database and the BRACE database
python3 brace-disassembler.py --db-root-2 brace-db/ -f ./bitstream.bits \
-o ./bitstream.fasm
```

Figure 3 - Creation and Enhancing BRACE's Database for Static Bitstream Checking

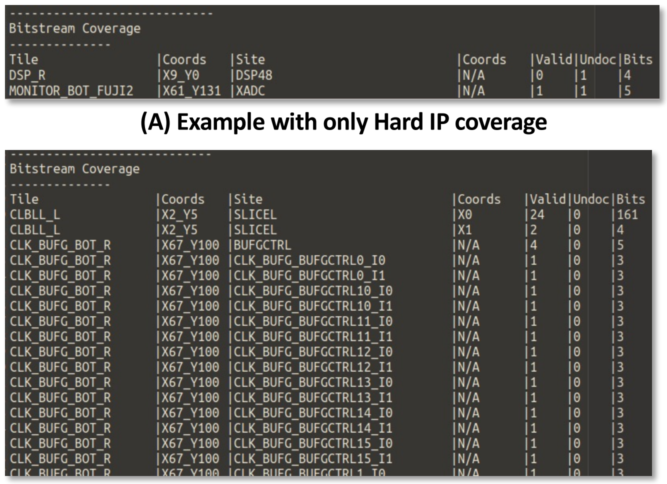
BRACE's configuration database is a superset of the known configuration settings from User's Guides and Application Notes. The database is generated for each part within a Xilinx FPGA family and combined with bitstream configuration settings extracted from open-source tools, such as Project X-Ray [6]. Each IP is analyzed for bitstream configuration coverage, and any missing bits or constrained parameters are evaluated to populate the *BRACE Enhanced Bitstream Database*, shown in Figure 3.

The BRACE Static Bitstream Checker generates a report that specifies the HARD IP in the design and if any undocumented bits or configuration values have been set. A high-level report is used to initially capture the status, seen in Figure 4(A) where a DSP and XADC IP block is reporting 1-bit each of undocumented state space. In Figure 4(B) the report is expanded to include all IP blocks for completeness in a design. Expansions of BRACE Reporting tools include the ability to pinpoint the bitstream bits for each undocumented state back to functionality, design documentation, or internal test cases that were used to derive the undocumented condition.

B. *BRACE Run-Time: Integrity Assurance Wrappers*

The need for run-time protection from undocumented functionality is an outgrowth of the user's, or an adversary's, ability to alter the configuration of FPGA Hard IP during runtime using control inputs or register-based interfaces such as the DRP. These interfaces create a vulnerability that could be enabled by data flowing through the device. To protect against these vulnerabilities, BRACE inserts run-time checkers known as Integrity Assurance Wrappers (IAW). These wrappers enclose FPGA Hard IP and monitor and sanitize control inputs that enable undocumented functions. In addition, alarm signals are output to inform high-level hardware (e.g. Xilinx Security Monitor) of attempts to enable undocumented functions. BRACE's Static bitstream checking is also used to ensure that wrappers are not accidentally removed or disabled by CAD-flow optimization or intentionally by bad actors.

BRACE uses a post-synthesis insertion process for the IAWs that is tightly integrated into the Xilinx tool flow



```
---------------------------
Bitstream Coverage
-------------
Tile                |Coords   |Site                                  |Coords    |Valid|Undoc|Bits
DSP_R               |X9_Y0    |DSP48                                 |N/A       |0    |1    |4
MONITOR_BOT_FUJI2   |X61_Y131 |XADC                                  |N/A       |1    |1    |5
```

(A) Example with only Hard IP coverage

```
---------------------------
Bitstream Coverage
-------------
Tile                |Coords   |Site                                  |Coords    |Valid|Undoc|Bits
CLBLL_L             |X2_Y5    |SLICEL                                |X0        |24   |0    |161
CLBLL_L             |X2_Y5    |SLICEL                                |X1        |2    |0    |4
CLK_BUFG_BOT_R      |X67_Y100 |BUFGCTRL                              |N/A       |4    |0    |5
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL0_I0                 |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL0_I1                 |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL10_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL10_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL11_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL11_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL12_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL12_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL13_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL13_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL14_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL14_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL15_I0                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL15_I1                |N/A       |1    |0    |3
CLK_BUFG_BOT_R      |X67_Y100 |CLK_BUFG_BUFGCTRL1_I0                 |N/A       |1    |0    |3
```

(B) Example with all coverage

Figure 4 - Output Reports from BRACE Static Bitstream Checker

(implemented using TCL scripts), seem in Figure 5. The insertion process first analyzes the settings of each supported IP and determines which runtime checks need to be actualized in hardware. In some cases, such as when control inputs are set to *valid* constant values, protection from undocumented functionality is inherent in the static configuration of the IP. This extra layer of analysis allows BRACE to provide maximum protection at a minimum hardware cost. The use of a post-synthesis insertion process allows the vendor place and route tools to ensure that the final design meets all timing requirements.

IAWs act as a filter for the runtime control inputs that passes valid inputs and suppresses inputs known to enable undocumented functionality. This filtering action allows the IAWs to preserve in the intended functionality of the device and remove undocumented inputs that cause vulnerabilities. The insertion process also creates an IP core that collects the alarm signals of *all* IAWs and provides and interface to higher-level hardware IP such as the Xilinx Security Monitor (SECMON),

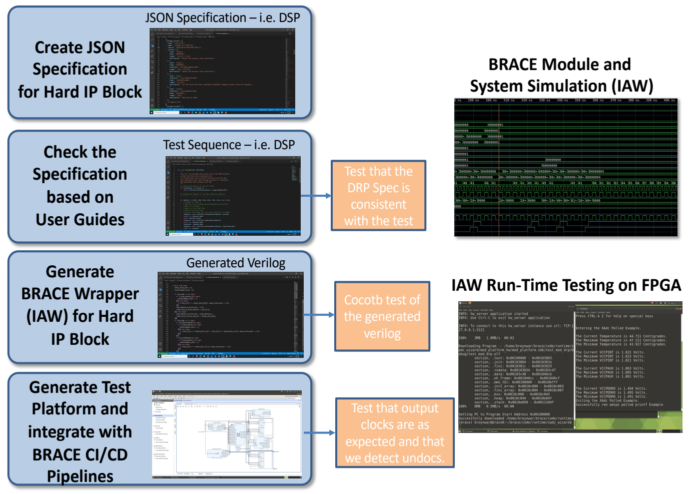

JSON Specification – i.e. DSP
Create JSON Specification for Hard IP Block
Test Sequence – i.e. DSP
Check the Specification based on User Guides
Test that the DRP Spec is consistent with the test
Generated Verilog
Generate BRACE Wrapper (IAW) for Hard IP Block
Cocotb test of the generated verilog
Generate Test Platform and integrate with BRACE CI/CD Pipelines
Test that output clocks are as expected and that we detect undocs.
BRACE Module and System Simulation (IAW)
IAW Run-Time Testing on FPGA

Figure 5 - BRACE Integrity Assurance Wrapper Development and Testing Framework

which then provides an interface that reports which alarm was triggered.

BRACE also provides a validation mechanism based on static bitstream checking and formal verification to ensure that all IAWs are unaltered prior the loading the bitstream on the device. Due to the complexities of the FPGA tool flows it is possible that hardware inserted by an out-of-band tool (such as BRACE) might be altered during the optimization process. It is also possible for bad actors and CAD tool flows to compromise the integrity of the IAW by intentionally altering the bitstream. The validation mechanism operates initially using an industry-standard formal verification tool (i.e. Synopsys Formality or Cadence Conformal) to assure the functional equivalence of the IAWs inserted into the post-synthesis netlist with those found in the post-place and route netlist. Using the post-place and route netlist, the bitstream mappings of wrapper primitives is identified and used to generate masks that are placed in the BRACE database. These masks can then be compared against the bitstream to ensure that the wrappers are still present. The result of this process is a light-weight static check that validates the IAWs and can be integrated with the static checker.

C. BRACE Validation and Testing Framework

BRACE seeks to deliver a robust prototype tool that can be rapidly transitioned into operational systems upon completion of the project. The goals of BRACE testing and validation efforts in this prototype tool are to 1) demonstrate the completeness and correctness of the BRACE database (for supported IP within the project scope), 2) demonstrate the accuracy and runtime performance of the static checker, 3) demonstrate the correctness and sensitivity of the Integrity Assurance Wrappers and 4) demonstrate that the software frameworks perform as expected and are reasonably resilient.

The accuracy and runtime of the BRACE static checker will be validated. The accuracy of the static checker will be validated by using the BRACE database and Automatic Test Pattern Generation (ATPG) to construct a large battery of test bitstreams containing undocumented functions. The test bitstreams will represent several devices from the main classes (i.e. SX, LX, TX) of Virtex-5 FPGAs. The validation process will report and check coverage metrics to ensure that *all* instances of undocumented functionality are tested across all devices in a sufficient number of locations on each device. Software development of the static checking software will use standard unit and integration testing practices to ensure the software is robust. The runtime complexity of the static checker will be evaluated during the testing process by ensuring that the test bitstreams represent a variety of devices including the largest of the Xilinx devices.

The filtering and alarms actions of the IAWs will be validated in RTL simulation to ensure that attempts to enable undocumented functionality are successfully filtered and reported (i.e. alarms are asserted). A constrained random approach will be used in the simulation testbench to ensure that all instances of undocumented functionality for supported IP are thoroughly tested. BRACE will further demonstrate the operation of IAWs in hardware by injecting undocumented control signal values into the IAWs and validate that these are indeed filtered and reported.

III. Conclusion

BRACE is an automated toolset that assures FPGA bitstreams do not activate instances of undocumented functionality that may result in security vulnerabilities. This is accomplished using both static bitstream checking and runtime Hard IP control checking during device operation. These checks are lightweight, operate within seconds to minutes, do not require a human analyst in the loop, and can be used in-situ in deployed embedded systems. By providing both static and runtime protection mechanisms, BRACE protects COTS FPGA bitstreams through their entire life cycles. BRACE is an open framework that can be extended to host other 3rd-party bitstream assurance analyses, such as automated wiring short detectors or other bitstream security tools.

IV. References

[1] B. Cohen, "IDA DoD Semiconductor Market Estimation," 2015

[2] M. French, A. Schmidt and A. Dasu, "Initial Approaches for Discovery of Undocumented Functionality in FPGAs," in *NDIA Trusted Microelectrons Conference: "Special Topic: Field Programmable Gate Array (fpga) Assurance"* , Mclean, VA, 2017.

[3] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas and M. French, "Torc: Tools for Open Reconfigurable Computing," in *19th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Salt Lake City, UT, USA, 2011.

[4] C. Lavin and A. Kaviani, "RapidWright: Enabling Custom Crafted Implementations for FPGAs," in *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Boulder, CO, USA, 2018.

[5] SymbiFlow. Project x-ray. https://github.com/ SymbiFlow/prjxray, 20