*Proc. of International Conference on Artificial Intelligence, Computer, Data Sciences and Applications (ACDSA 2025)*
*7-9 August 2025, Antalya-Türkiye*

# "vcd2df" - Leveraging Data Science Insights for Hardware Security Research

Calvin Deutschbein
*School of Computing and Information Sciences*
*Willamette University*
Oregon, United States of America
0000-0003-1354-7200

Jimmy Ostler
*School of Computing and Information Sciences*
*Willamette University*
Oregon, United States of America
jtostler@willamette.edu

***Abstract*—In this work, we hope to expand the universe of security practitioners of open-source hardware by creating a bridge from hardware design languages (HDLs) to data science languages like Python and R through libraries that converge VCD (value change dump) files into data frames, the expected input type of the modern data science tools. We show how insights can be derived in high-level languages from register transfer level (RTL) trace data. Additional, we show a promising future direction in hardware security research leveraging the parallelism of the Spark DataFrame.**

***Index Terms*—Hardware, Security, Machine learning, Data science, Open source, Verilog, HDL, RTL, Python, Spark, R, Data frames.**

## I. Introduction

What is a "vcd", or value change dump? "[A]n ASCII-based format for dumpfiles generated by EDA logic simulation tools. The standard, four-value VCD format was defined along with the Verilog hardware description language by the IEEE Standard 1364-1995 [1] in 1996."[1]. VCD files capture traces of execution of simulated software design, for which hardware engineers often interact with the traces through domain-specific tools like waveform viewers, such as GTKWave[2].

What is a "df", or data frame? "[O]ne of the most common data structures used in modern data analytics because they are a flexible and intuitive way of storing and working with data."[3]. Each year, in the U.S. alone, hundreds of thousands of students learn about data frames in computer science, data science, and scientific computing courses and courses in allied disciplines such as the physical sciences and economics. The overwhelming popularity of these tools is driven by their broad usefulness and accessibility.

Modern hardware designs are exceedingly complex [28], on the order of 10 billion transistors for consumer CPUs (central processing unit). To combat complexity, Hardware Description Languages (HDLs) enable hardware designers to design software by writing code. To become still more accessible, we can recognize that traces of execution of a hardware design are no different than any other observations from a statistical or data scientific perspective and can be interacted with via the same methodologies.

In this work, we will introduce "vcd2df" libraries for what we believe to be the three most common data science frameworks: Python "pandas", ASF Spark, and the R Project for Statistical Computing, of which ASF Spark incidentally supports Python and R frontends. For each, we will show an example of a hardware design, a trace as VCD file, and an insight we can discover with language built-in functions and methods. We recognize that a simulation-only approach is insufficient for some hardware goals but still supports hardware quality assurance.

1) **The R Project**: A "free software environment for statistical computing and graphics."[4]
2) **Python "pandas"**: A "fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language."[5]
3) **ASF Spark**: A "multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters."[6]

## II. Background

We briefly introduce the input and output types of our framework: (1) HDLs and the VCD file type, and (2) Data frames.

### A. HDLs and the VCD file type

VCD files are generated by software tools which simulate, emulate, or compile in a machine-executable language some hardware design, expressed in a hardware design language such as Verilog or VHDL. They commonly have three primary inputs: design, testbench, and simulator.

While many aspects of the VCD file format are unimportant, we do note that the term "value change" refers to the fact that

This work was supported by NSF Award 171858

[1]https://en.wikipedia.org/wiki/Value_change_dump
[2]https://gtkwave.sourceforge.net/
[3]https://www.databricks.com/glossary/what-are-dataframes
[4]https://www.r-project.org/
[5]https://pandas.pydata.org/
[6]https://spark.apache.org/

VCD files only log changes to the bits within some register (what are essentially a fixed number of fixed size variables in a hardware design), which is commonly expressed as binary numeric data, though VCD files allow hardware signals in bits in addition to '0' and '1', so a bit may have a value of 'x' or 'z'. This merited three major design decisions for these packages:

1) First, we treat any register containing any non-numeric value as having a value of '-1', which is easy to encode in all data frame implementations and not a valid register value, as register values are expressed as natural numbers in binary.
2) Second, we implement our packages via file streaming, rather than loading the entire file at once into memory, as we necessarily maintain prior register values in the data frame and may read a single value change at a time.
3) Third, we noted that many VCD files contained registers of the same name in different modules, a hardware abstraction similar to the software abstraction of the same name, that never differed in value, and removed these to avoid data duplication.

Together, we believe these design decisions balance correctness and performance.

*1) A Design:* The primary input is a hardware design specified in an HDL such as Verilog or VHDL. In general, we expect a design specified at the register transfer level (RTL). In the case of Verilog, these are often specific with a ".v" suffix and specify hardware designs as a series of registers or signals joined by wires. This represents the digital logic of some hardware design, but can readily be interpreted as a program or executable.

There are numerous open-source Verilog designs made available through a wide-range of research [27] and industry [24] projects.

*2) A Testbench:* An HDL description of hardware cannot be executed and therefore cannot generate a trace of execution, which is necessary to create a trace of execution and have values which to log. So VCD generation also requires a testbench which in some way exercises the hardware design, usually through a series of loops and inputs. Testbench generation is a separate and active area of research [35], but we only require that some imperatives be dispatched to a hardware design from a simulation framework that may log the hardware state.

In general, we find that testbenches are often maintained under version control in the same HDL as the design to which they accompany, as development without testbenches is exceedingly difficult and uncommon. For all designs we explored, we were able to use existing testbenches written by project maintainers.

*3) A Simulator:* We find that the most commonly recommended tool for scholarship on hardware design is Icarus Verilog, a free and open-source Verilog compiler under the GPL license, maintained on GitHub[7]. "Icarus Verilog is not aimed at being a simulator in the traditional sense, but a compiler that generates code employed by back-end tools." Icarus Verilog supports the VCD file as a default output type.

[7]https://github.com/steveicarus/iverilog

We should note that the VCD format is ASCII-based format, incurring write-speed limits at one-eighth of the speed of a binary representation (when printing textual binary, one bit of information incurs eight bits of storage). In initial experiments, we have found it possible to bypass the VCD representation and stream directly into a data science framework, but this removes the benefit of encapsulating hardware design complexities entirely. We hope to explore streaming data during hardware simulation in future work.

## *B. Data Frames*

Rather than being the result of an IEEE standard, the notion of a data frame emerged naturally over time as statistical, scientific computing, and data analytics libraries became more mature in scripting languages and cloud computing applications. We are aware of no precise definition or history around data frames, but think of them as a base feature the R Project, which was formally launched in 1993, and incorporated into Python by "pandas" (for "panel data") in 2008 and then by Spark 1.6 in 2015. Statisticians developing R noted the usefulness of storing data with with variables in columns and cases, or observations, in rows. In 2025, "pandas" (and its dependency NumPy) are the two most popular libraries in the most popular programming language (Python) and Spark is one of the most popular platforms for distributed computing.

We formally regard a data frame as a "tabular data structure common to many data processing libraries"[8] and think of them as fulfilling many of the goals of spreadsheet applications with the benefits of the data structure being accessible as an object within a scripting environment. It is useful as a technology to scale data analysis beyond graphical user interfaces and facilitate automation.

*1) The R Project:* The R Project uses the term "data frame" to refer to lists of the "data.frame" class, and (following arrays, matrices, and lists) is the most basic data structure in R which is not common to other programming languages. Essentially, they are implemented as lists of vectors, which may be lists, array, martices or other data frames. These internal vectors must all be of the same length, as in a matrix.

In R, the data frame was most initially used most commonly for reading data from files, and expects what are essentially separated value files with support for row and column labels.

Many R packages work directly on data frames for statistical work as well as machine learning techniques such as clustering and classification.

*2) Python "pandas":* The "pandas" library regards the DataFrame as its primary data structure. The DataFrame is implemented as a dictionary, or map, like structure over "Series" objects, its vector type. The vectors are implemented as NumPy arrays, which are in turn implemented as arrays in the C programming language.

The usefulness of this implementation for using data frames to analysis hardware designs is that same underlying hardware

[8]https://en.wikipedia.org/wiki/Dataframe

being studied, which follows many rules such as storing integers in registers of bytes, can be used to analyze those same technologies when used to implement other (or even the same) hardware designs.

To our knowledge, “pandas” has the most users of any data frame or even any script- or programming-based technology for data analysis. However, it is not necessarily the largest in terms of data analyzed.

*3) ASF Spark:* Spark and its underlying insight, resilient distributed datasets (RDDs) [33] was introduced in 2014 to address limitations of the earlier paradigm of MapReduce [10] and was an important innovation in distributed computing, but quickly also made important contributions to the data frame formulation with the launch of the DataFrame API over RDDs, which is now supported in Spark via Python, R, Scala, and Java interfaces.

Spark provides only benefits and no real costs to development of Python and R analytics libraries. With “vcd2df” functions in either language, existing Spark APIs allow reading Python and R data frames into internal Spark distributed computing and provide additional capabilities without any additional development costs to the package maintainers.

A core motivation for this work was to develop frameworks that scale to any hardware designs, including CISC (Complex Instruction Set Computer) CPUs, SoCs (System-on-a-Chip), and other hardware that would be impractical to simulate and analyze on a single device. For example, automated testing for transient execution CPU vulnerabilities (such as Spectre [21] or Meltdown [23]) is a long-standing goal of hardware security research [11] but requires extensive computation. We hope to explore the usage of Spark for this application in future work.

## III. Implementation

A variety of existing tools interact with VCD files, yet to our knowledge no currently supported tools read VCD files into data frames of any kind. However, we were able to adapt work from the Mythra[9] package for containerized hardware CI/CD, which used a custom Python script to parse VCD files into other file types, and instead target data frame output. We implemented the VCD-to-data-frame parsing natively in both Python, with a “pandas” dependency, and in R with no dependencies, so that the package may be easily maintained and distributed.

Regarding native implementation versus the use of a higher performance low-level language such as C/C++ or Rust, we expect a performance bottleneck on HDL simulation rather than on our parsing for all design. However, if performance in these libraries becomes a bottleneck on design, we expect the most likely improvements to come from streaming value changes with no intermediate write-to-disk or ASCII encoding, which we hope to explore in a future work.

Both packages provide the 'vcd2df' function, which loads a IEEE 1364-1995/2001 VCD (.vcd) file, specified as an parameter as a string containing exactly a file path, and returns ether a pandas DataFrame or an R Project list of class data.frame containing values over time. A VCD file captures the register values at discrete timepoints from a simulated trace of execution of a hard ware design in Verilog or VHDL. The returned dataframe contains a row for each register, by name, and a column for each time point, specified VCD-style using octothorpe-prefixed multiples of the timescale.

*1) The R Project:* As part of research efforts, we have submitted to CRAN (The Comprehensive R Archive Network) a “vcd2df” package[10] which implements a single function of the same name. We implemented the package natively in R in 86 lines-of-code, of which 33 were comments and 3 were whitespace only.

*2) Python “pandas”:* As part of research efforts, we have published to PyPI (The Python Package Index) a “vcd2df” package[11] which implements a single function of the same name. We implemented the package natively in Python in 36 lines-of-code, of which 3 were comments and 3 were whitespace only. We may use this same package in ASF Spark via the PySpark interface, or use this package to write Parquet files for direct usage in Spark.

## IV. Case Studies

We tested over three distinct hardware designs, using one technology per design. We used the demonstration designs for the Mythra package which include two RISC-V [3] Reduced Instruction Set Computers (RISC) and an access control module [27]:

1) NERV - Naive Educational RISC-V Processor[12]
2) PicoRV32 - A Size-Optimized RISC-V CPU[13]
3) AKER-Access-Control[14]

Each design was written in Verilog, simulated with Icarus Verilog, and open-sourced on GitHub. We used Icarus Verilog version 11 rather than the most recent release (12) to avoid a versioning issue with the latest Verilog standard and some non-standard implementation features of NERV. While these files can be generated from open source designs with open source tools, we also maintain a repository with the VCD files[15] for accessibility and reproducibility.

We timed translations via the ‘time’ command and the average and standard deviation from 10 translations, after writing minimal scripts that wrapped the “vcd2df” library import and function call.

In R, we wrote:

```
library(vcd2df)

if (!interactive()) {
    args <- commandArgs()
    f_name <- args[length(args)]
```

[9]https://github.com/hwcicd/myrtha

[10]https://github.com/vcd2df/vcd2df

[11]https://pypi.org/project/vcd2df/1.0/

[12]https://github.com/YosysHQ/nerv

[13]https://github.com/YosysHQ/picorv32

[14]https://github.com/KastnerRG/AKER-Access-Control

[15]https://github.com/vcd2df/vcd_ex

```
    df <- vcd2df(paste0(f_name, ".vcd"))
    saveRDS(df, paste0(f_name, ".rds"))
}
```

In Python, the "pandas" I/O API supports many filetypes, including both text-based and binary files, and with the addition of "pyreadr" module can also write to the R Project RDS filetype. We wrote:

```
from vcd2df import vcd2df
# import pyreadr

if __name__ == "__main__":
    import sys
    f_name = sys.argv[1]
    df = vcd2df(f_name+".vcd")
    df.to_pickle(f_name+".pkl")
    # df.to_parquet(f_name+".parquet")
    # df.to_feather(f_name+".ftr")
    # pyreadr.write_rds(f_name+".rds", df)
```

We should note we tested both the "pyarrow" and "fastparquet" engines for Parquet files. Parquet and Feather are ASF file types and well suited to Spark (and usable in R packages with appropriate R packages), and "pickle" is a Python file type, essentially the Python equivalent of the R Project RDS file type.

## A. NERV

NERV contains 549 registers in 1267 lines of SystemVerilog. Its accompanying testbench is 155 SystemVerilog LoC and runs for 19 cycles. In R, translation took an average of .194 seconds with a standard deviation of .005 seconds. In Python, translation to "pickle" took an average of .303 seconds with a standard deviation of .006 seconds. We report the file sizes and savings ratios in Tab. I. We remark briefly on the space savings, for example, the VCD file was 31.0 kilobytes in size but the resultant RDS file was only 2.51 kilobytes in size, a space savings of 93%.

| File Type | File Ext. | Engine | Bytes | Save% | ms |
|---|---|---|---|---|---|
| VCD | .vcd | iverilog | 31049 | - | - |
| RDS | .rds | R | 2512 | 93.0 | 194 |
| pickle | .pkl | pandas | 35926 | -8.72 | 312 |
| Parquet | .parquet | PyArrow | 8629 | 76.0 | 352 |
| Parquet | .parquet | Fast | 6646 | 81.5 | 309 |
| Feather | .ftr | PyArrow | 9786 | 72.8 | 318 |
| RDS | .rds | pyreadr | 30057 | 16.3 | 318 |

TABLE I
NERV DATA FRAME SIZES

To show the usefulness of data frames for studying NERV traces, we note that the space savings can be unexpectedly high, so we may wish to assess the coverage of our testbench. We can do so with a few simple lines of R:

```
> library(vcd2df)
> nerv <- vcd2df("nerv.vcd")
> count <- data.frame(rowSums(nerv > 0))
> colnames(count) <- "counts"
> rownames(subset(count, counts > 0))
```

We can quickly see that only 20 registers (of 549) are initialized and set to non-zero values in this brief test, and can additionally see their names, which includes noteworthy registers such as 'trap', 'clock', and 'reset', common classes of names for registers defining the control state of same CPU design such as NERV, but notably does not include 'pc', the program counter, usually the most important register for denoting progression through some trace of execution as it traces successive instructions read from memory.

## B. PicoRV32

PicoRV32 contains 232 registers in 3049 lines of Verilog code. Its accompanying testbench is 86 lines of Verilog and runs for 2201 cycles. In R, translation took an average of 1.618 seconds with a standard deviation of .024 seconds. In Python, translation to "pickle" took an average of .601 seconds with a standard deviation of .047 seconds. We report the file sizes and savings ratios in Tab. II. We remark briefly on the space cost, for example, the VCD file was 269 kilobytes in size but the resultant "pickle" file was 3963 kilobytes in size, a space cost of 14.6×.

| File Type | File Ext. | Engine | Bytes | Size× | ms |
|---|---|---|---|---|---|
| VCD | .vcd | iverilog | 269184 | - | - |
| RDS | .rds | R | 107012 | .398 | 1618 |
| pickle | .pkl | pandas | 3917766 | 14.6 | 601 |
| Parquet | .parquet | PyArrow | 1704664 | 6.33 | 706 |
| Parquet | .parquet | Fast | 1191528 | 4.42 | 1533 |
| Feather | .ftr | PyArrow | 1717122 | 6.37 | 681 |
| RDS | .rds | pyreadr | 3962508 | 14.7 | 1150 |

TABLE II
PICORV32 DATA FRAME SIZES

To show the usefulness of data frames for studying PicoRV32 traces, we note that the "32" in name stands for the word size, and we can examine register to see if the contain what we expect to be memory addresses - values smaller than $2^{32}$ which are also multiples of 32. We term this set $M$.

$$M = \{m \in \mathbb{N} : (32 \mid m) \wedge (m < 2^{32})\}$$

It is trivial to implement the membership test in Python, both for individual values and for collections of values, as either functions or lambda expressions containing the following code blocks:

```
not (m % 32) and (m < (1 << 32)) # single
all([f(m) for m in ms]) # collection
```

From there, we construct a series of registers for which this predicate holds at all time points.

```
df = pd.read_pickle("pico.pkl")
df.apply(f_all, axis=1)
```

We can quickly see that only 56 registers (of 221) are 32 bit values which are mulitples of 32 at all time points, including some memory registers such as 'mem_la_firstword'. However, not all 'mem' prefixed registers, like 'mem_wstrb', are in this series, suggesting we should further study our assumptions of the PicoRV32 implementation.

### *C. AKER*

AKER contains 432 registers in two files totaling 2002 Verilog LoC. Its accompanying testbench is 527 lines of SystemVerilog and runs for 1055 cycles. In R, translation took an average of .423 seconds with a standard deviation of .010 seconds. In Python, translation to "pickle" took an average of .440 seconds with a standard deviation of .007 seconds. We report the file sizes and savings ratios in Tab. III.

| File Type | File Ext. | Engine | Bytes | Size× | ms |
|---|---|---|---|---|---|
| VCD | .vcd | iverilog | 53007 | - | - |
| RDS | .rds | R | 56969 | 1.07 | 423 |
| pickle | .pkl | pandas | 2977387 | 56.1 | 440 |
| Parquet | .parquet | PyArrow | 888735 | 16.8 | 518 |
| Parquet | .parquet | Fast | 684618 | 12.9 | 949 |
| Feather | .ftr | PyArrow | 952394 | 18.0 | 489 |
| RDS | .rds | pyreadr | 2997098 | 56.5 | 859 |

TABLE III
AKER DATA FRAME SIZES

For AKER, to exercise Spark, we use 229 VCD files generated by the Isadora tool for automated information flow property generation for hardware designs [11], which open-sourced its evaluation data via GitHub[16]. Each VCD file tracks potential sensitive information from one for the 229 registers and denotes an information flow into another register by setting a register of the same name prefixed with "shadow_" to a non-zero value. We slightly reworked the "vcd2df" function to read strings as "strd2df", rather than files, and read the VCD files into a Spark Dataframe as text data then convert to a "pandas" DataFrame.

```
vcds = "gs://vcds-aker/vcds/*.vcd"
df = spark.read.text(vcds, wholetext=True)
rdd = df.rdd.map(lambda x : str2df(x[0]))
```

From there, we could straightforwardly inspect "shadow" registers, registers which denote information flow tracking information, to find the first non-zero instance (when sensitive information may reach a register).

```
def iflow_times(x):
    x = x[x.index.str.contains("shadow")]
    x = x[x.any(axis=1)]
    x = x.idxmax(axis=1)
    x = x[x != "#0"]
    return x
```

We believe the the resultant DataFrame contains original research in the hardware security space, which regard as out of scope for this publication but hope to explore in greater detail in future work. We have made the the Spark script available via GitHub Gist[17].

## V. Related Work

*Data Scientific Approaches to Hardware:* Early computational methods to assess the correctness and security of hardware designs often directly examined trace data for known patterns, such as one-hot encoding [14], [18]. Other researchers applied data mining [6], [19], [22] or temporal logic [7]–[9], [12] techniques. Specific the to security context, researchers have manually identified properties [4], [5], [20] and used software tools incorporating clustering and classification for RISC [34] and CISC [13] designs. Recent work has discovered subsets of hyperproperties [11], [25], which are defined as relationships over multiple traces. In this work, we use Spark to evaluate multiple traces of a single design (PicoRV32) to discover grammatically similar hyperproperties to these works.

*Data Scientific Approaches to Correctness:* Ammons et al. introduced specification mining [2], a form of data mining for computing systems by considering their traces of execution, and did so in a software context. Specification mining launched a rich research direction across static and dynamic analysis [31]; imperfect traces [32]; and complex races [16], [17], [26]. Perhaps the most widely known miner, Daikon [15] approached specification mining as invariant detection. An inspiration for this work was studying the Daikon source code and seeing custom implementations of $k$-means and hierarchical clustering used within its heuristics for determining program correctness. We regard the Python and R implementations of these algorithms as the state-of-the-art and wished to apply them directly to hardware traces.

*Python Testbenches:* The use of languages common to data analytics for hardware evaluation is not novel, with perhaps the most common example being specifying testbenchs in Python [29] which allows study of hardware design while abstracting register transfer level notions of sequential and combinatorial logic and non-numerical register values.

Pyverilog [30] was an earlier effort to bring Verilog tooling, such as code generation and code flow analysis, into a higher-level language.

We believe both of these contributions take advantage of the accessibility of Python as a scripting language, but do not leverage its unique capabilities for data analysis.

## VI. Conclusion

We have proposed and demonstrated "vcd2df", a proof-of-concept package for studying hardware designs with existing data science techniques. We have distributed our package through the relevant package manages for Python and the R Project, and have shown some simple examples of how easily insights may be learned from data frames containing trace data.

## Acknowledgements

We thank the NSF for funding. We thank the CRAN network for their helpful comments while developing "vcd2df" for the R Project. We thank Andres Meza for the AKER testbench. We thank Intel Corporation, the Kastner Research Group, and the HWSec@UNC research group for helpful discussions. We thank our undergraduate researchers, Kendall Leonard and Hannah Pahama, for suggesting library support

[16]https://github.com/cd-public/Isadora/tree/master/model/single/vcds
[17]https://gist.github.com/cd-public/6dd2db85d2bbcdb623315b8f56906450

for interfacing with VCD files during their research experience in the summer of 2024.

## References

[1] Ieee standard verilog hardware description language. *IEEE Std 1364-2001*, pages 1–792, 2001.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[3] K. Asanović and D. A. Patterson. Instruction sets should be free: the case for RISC-V. Technical Report UCB/EECS-2014-146, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA, August 2014.

[4] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 34–39. IEEE, June 2011.

[5] Michael Brown. Cross-validation processor specifications. Master's thesis, University of North Carolina at Chapel Hill, 2017.

[6] Po-Hsien Chang and Li C Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 607–612. IEEE, 2010.

[7] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 67–72, March 2015.

[8] A. Danese, G. Pravadelli, and I. Zandonà. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 606–611, March 2016.

[9] A. Danese, N. D. Riva, and G. Pravadelli. A-TEAM: Automatic template-based assertion miner. In *Proceedings of the 54th Design Automation Conference (DAC)*, pages 1–6. ACM/EDAC/IEEE, June 2017.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[11] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. Isadora: Automated information flow property generation for hardware designs. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*, ASHES '21, page 5–15, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Calvin Deutschbein and Cynthia Sturton. Mining security critical linear temporal logic specifications for processors. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 18–23, 2018.

[13] Calvin Deutschbein and Cynthia Sturton. Evaluating security specification mining for a cisc architecture. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 164–175, 2020.

[14] E. El Mandouh and A. G. Wassal. Automatic generation of hardware design properties from simulation traces. In *International Symposium on Circuits and Systems (ISCAS)*, pages 2317–2320. IEEE, 2012.

[15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[16] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE)*, pages 339–349. ACM, 2008.

[17] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 51–60. ACM, 2008.

[18] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of 42nd Design Automation Conference (DAC)*. IEEE, 2005.

[19] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[20] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 517–529. ACM, 2015.

[21] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[22] Wenchao Li, Alessandro Forin, and Sanjit A Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference*, pages 755–760, 2010.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[24] Emanuele Parisi, Alberto Musa, Maicol Ciani, Francesco Barchi, Davide Rossi, Andrea Bartolini, and Andrea Acquaviva. Assessing the performance of opentitan as cryptographic accelerator in secure open-hardware system-on-chips, 2024.

[25] Mayank Rawat, Sujit Kumar Muduli, and Pramod Subramanyan. Mining hyperproperties from behavioral traces. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 88–93, 2020.

[26] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *28th International Conference on Automated Software Engineering (ASE)*, pages 658–663. IEEE/ACM, 2013.

[27] Francesco Restuccia, Andres Meza, and Ryan Kastner. Aker: A design and verification framework for safe and secure soc access control. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[28] Martin Schoeberl. Open-source hardware design.

[29] Varun Sharma, Naif Tarafdar, and Paul Chow. Sonar: Writing testbenches through python. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 311–311, 2019.

[30] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[31] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476. Springer-Verlag, 2005.

[32] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 282–291. ACM, 2006.

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.

[34] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. Identifying security critical properties for the dynamic verification of a processor. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 541–554. ACM, 2017.

[35] Ziyue Zheng, Xiangchen Meng, and Yangdi Lyu. Ape-fv: Concolic testing for rtl functional verification using adaptive path exploration. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 373–380, 2024.