Hardware/Software Co-Assurance for the Rust Programming Language Applied to Zero Trust Architecture Development

David Hardin
Collins Aerospace
Cedar Rapids, Iowa, USA
david.hardin@collins.com

ABSTRACT

Zero Trust Architecture requirements are of increasing importance in critical systems development. Zero trust tenets hold that no implicit trust be granted to assets based on their physical or network location. Zero Trust development focuses on authentication, authorization, and shrinking implicit trust zones to the most granular level possible, while maintaining availability and minimizing authentication latency. Performant, high-assurance cryptographic primitives are thus central to successfully realizing a Zero Trust Architecture. The Rust programming language has garnered significant interest and use as a modern, type-safe, memory-safe, and potentially formally analyzable programming language. Our interest in Rust particularly stems from its potential as a hardware/software co-assurance language for developing Zero Trust Architectures. We describe a novel environment enabling Rust to be used as a High-Level Synthesis (HLS) language, suitable for secure and performant Zero Trust application development. Many incumbent HLS languages are a subset of C, and inherit many of the well-known security shortcomings of that language. A Rust-based HLS brings a single modern, type-safe, memory-safe, high-assurance development language for both hardware and software. To study the benefits of this approach, we crafted a Rust HLS subset, and developed a frontend to the hardware/software co-assurance toolchain due to Russinoff and colleagues at Arm, used primarily for floating-point hardware formal verification. This allows us to leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. In this paper, we describe our Rust subset, detail our prototype toolchain, and describe the implementation, performance analysis, formal verification and validation of representative Zero Trust algorithms and data structures written in Rust, emphasizing cryptographic primitives and common data structures.

KEYWORDS

hardware/software co-assurance, Rust, theorem proving, ACL2

ACM Reference Format:
David Hardin. 2022. Hardware/Software Co-Assurance for the Rust Programming Language Applied to Zero Trust Architecture Development. In *Proceedings of (Ada Letters).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

Ada Letters, December 2022, ACM

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Zero Trust Architecture [28] requirements are increasingly becoming adopted in critical systems development. Zero trust tenets state that there is no implicit trust granted to assets based on their physical or network location. All communications should be conducted "in the most secure manner available, protect confidentiality and integrity, and provide source authentication" [28]. Zero trust architectures shrink implicit trust zones to the most granular level possible, while maintaining availability and minimizing authentication delays. Performant, high-assurance cryptographic technologies are thus central to successfully realizing a Zero Trust Architecture, as are "leak-free" data structures, and other high-assurance components.

We have developed several zero trust primitives as part of the DARPA CASE program [6]. As the Zero Trust Architecture specification [28] was not created until the CASE program was well underway, this was not an explicit goal of the program, but similar cyber-assurance concerns informed both efforts, resulting in convergent technologies. A major guiding principle of CASE is the need for verified and validated automated synthesis of security-enhancing components from high-level architectural specifications, including input filters [12], safety monitors [22], remote attestation and measurement [26], as well as trustworthy interprocess communications [14]. Our research and development effort on CASE has emphasized the value of modern, type-safe, memory-safe, and formally analyzable languages for use in automated synthesis [12, 14, 26], and has identified the value of automated high-assurance synthesis to hardware, software, or a combination of the two, from architectural level specifications [10].

In this paper, we describe the development, formal verification, and validation of a number of zero trust architecture primitives in a High-Level Synthesis (HLS) subset of Rust, suitable for software and/or hardware implementation. Along the way, we introduce Rust, outline our HLS subset, describe a prototype hardware/software co-assurance toolchain for this subset, present case studies of zero trust primitives, and detail verification and validation efforts. It is hoped that this explication will convince the reader of the practicality of Rust as a high-assurance hardware/software co-design language, as well as the feasibility of performing full functional correctness proofs of code written in this Rust subset. We describe related work, then provide concluding remarks.

2 THE RUST PROGRAMMING LANGUAGE

The Rust Programming Language [16] is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. A distinguishing feature of Rust is a non-scalar object may only have one owner. For example, one cannot assign a reference to an object in a local variable, and then pass that reference to a function. The Rust runtime performs array bounds checking, as well as arithmetic overflow checking (the latter can be disabled by a build environment setting). In most other ways, Rust is a fairly conventional modern programming language, with interfaces (called traits), lambdas (termed closures), and pattern matching, as well as a macro capability. Also in keeping with other modern programming language ecosystems, Rust features a build and package management system, named cargo.

Rust has garnered significant interest and use as a modern, type-safe, memory-safe language, with compiled code performance approaching that of C/C++. Google [31] and Amazon [24] make significant use of Rust, and Linus Torvalds has commented positively on the near-term ability of the Rust toolchain to be used in Linux kernel development [1]. The latter capability comes none too soon, as use of C/C++ continues to spawn a seemingly never-ending parade of security vulnerabilities, which continue to manifest at a high rate [23] despite the emergence and use of sophisticated C/C++ analysis tools.

Our interest in Rust additionally stems from its (until now, unrealized) potential as a hardware/software co-assurance language that can be used to create high-assurance systems, including those that must meet zero trust architecture requirements. We are particularly motivated by new autonomous and semi-autonomous platforms that require sophisticated algorithms and data structures, are subject to stringent accreditation/certification, and encourage hardware/software co-design approaches. (For an unmanned aerial vehicle use case illustrating a formal methods-based systems engineering environment, please consult [22].) In this paper, we explore the use of Rust as a High-Level Synthesis (HLS) language [25]. Most incumbent HLS languages are a subset of C, e.g. Mentor Graphics' Algorithmic C [20], or Vivado HLS by Xilinx [33]. A Rust-based HLS would bring a single modern, type-safe, and memory-safe expression language for both hardware and software realizations, with very high assurance.

Another keen research topic is reasoning about application logic written in the imperative style favored by industry. Much progress has been made in this area in recent years, and we can now verify the correctness of algorithm and data structure code that utilizes idioms such as records, loops, modular integers, and the like; and verified compilers can guarantee that such code is compiled correctly to binary [17, 18]. Progress has also been made in the verification of hardware/software co-design algorithms, where array-backed data structures are common [9, 10]. (NB: This style of programming addresses one of the shortcomings of Rust, namely its lack of support for cyclic data structures.)

3 HARDWARE/SOFTWARE CO-ASSURANCE AT SCALE

In order to begin to realize our aspirational vision for hardware/software co-assurance at scale, we have conducted several experiments employing a state-of-the-art toolchain, due to Russinoff and O'Leary, and originally designed for use in floating-point hardware verification [29], to determine its suitability for the creation of safety-critical/security-critical applications in various domains.

Algorithmic C [20] is a High-Level Synthesis (HLS) language, and is supported by hardware/software co-design environments from Mentor Graphics, *e.g.*, Catapult [21]. Algorithmic C defines C++ header files that enable compilation to both hardware and software platforms, including support for the peculiar bit widths employed, for example, in floating-point hardware design. Restricted Algorithmic C (RAC) imposes several restrictions beyond those of Algorithmic C. The most significant of these is that pointers are not allowed, all loops must terminate, and all functions must be side-effect-free.

An ACL2 translator converts imperative RAC code to functional ACL2 code. Loops are translated into tail-recursive functions, with automatic generation of measure functions to guarantee admission into the logic of ACL2 (RAC subsetting rules ensure that loop measures can be automatically determined). Structs and arrays are converted into functional ACL2 records. The combination of modular arithmetic and bit-vector operations of typical RAC source code is faithfully translated to functions supported by Russinoff's RTL theorem library. ACL2 is able to reason about non-linear arithmetic functions, so this usual concern is not an issue. Finally, the RTL theorem library in ACL2 is capable of reasoning about a combination of arithmetic and bit-vector operations, which is a very difficult feat for most automated solvers.

Recently, we have investigated the synthesis of Field-Programmable Gate Array (FPGA) hardware directly from high-level architecture models, in collaboration with colleagues at Kansas State University. The goal of this work is to enable the generation of high-assurance hardware and/or software from high-level architectural specifications expressed in the Architecture Analysis and Design Language (AADL) [8], with proofs of correctness in ACL2.

4 RESTRICTED ALGORITHMIC RUST

As a study of the suitability of Rust as an HLS, we have crafted a Rust subset, inspired by RAC, which we have imaginatively named Restricted Algorithmic Rust, or RAR [11]. In fact, in our first implementation of a RAR toolchain, we merely "transpile" (perform a source-to-source translation of) the RAR source into RAC. By so doing, we leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. By transpiling RAR to RAC, we gain access to existing HLS compilers (we can generate code for either the Algorithmic C or Vivado HLS toolchains via some simple C preprocessor directives). We are also able to leverage the RAC-to-ACL2 translator that Russinoff and colleagues at Arm have successfully utilized in industrial-strength floating point hardware verification.

As we wish to utilize the RAC toolchain as a backend in our initial work, we adopt the same semantic restrictions for RAR as described in Russinoff's book. Additionally, in order to ease the

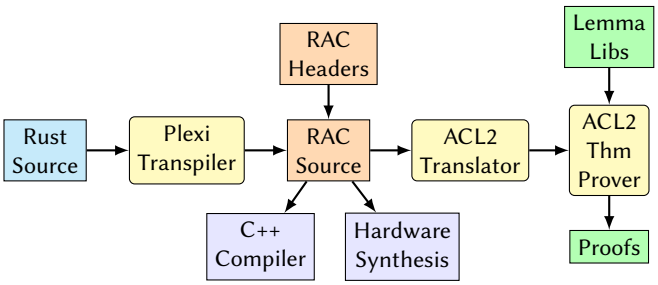

Figure 1: Restricted Algorithmic Rust (RAR) prototype toolchain.

transition to/from C, we support a commonly used macro that provides a C-like *for* loop in Rust. Note that, despite the restrictions, RAR code is proper Rust; it compiles to binary using the standard Rust compiler.

RAR is transpiled to RAC via a source-to-source translator, as depicted in Fig. 1. Our transpiler is based on the *plex* parser and lexer generator [30] source code. We thus call our transpiler *Plexi*, a nickname given to a famous (and now highly sought-after) line of Marshall guitar amplifiers of the mid-1960s. Plexi performs lexical and syntactic transformations that convert RAR code to RAC code. This RAC code can then be compiled using a C/C++ compiler, fed to an HLS-based FPGA compiler, as well as translated to ACL2 via the RAC ACL2 translator, as illustrated in Fig. 1.

We have implemented several representative algorithms and data structures in RAR, including:

- a suite of array-backed algebraic data types, previously implemented in RAC [9, 11];
- a significant subset of the Monocypher [32] modern cryptography suite, including XChacha20 and Poly1305 (RFC 8439) encryption/decryption, Blake2b hashing, and X25519 public key cryptography; and
- a DFA-based JSON lexer, coupled with an LL(1) JSON parser. The JSON parser has also been implemented using Greibach Normal Form (previously implemented in RAC, as described in [12]).

The RAR examples created to date are similar to their RAC counterparts in terms of expressiveness, and we deem the RAR versions somewhat superior in terms of readability (granted, this is a very subjective evaluation). Additionally, RAR support of basic Rust syntax gives embedded developers an "on-ramp" to a more modern development language, as opposed to being forced to stay with C in order to achieve high performance and low latency. Further, the benefits of using the Rust compiler in RAR code development should not be discounted: its enforcement of type safety and memory safety, coupled with its efficient code generation capability, encourages performant, high-quality code development.

5 EXAMPLES

5.1 Circular Queue

High-assurance data structures are necessary building blocks for any zero-trust architecture realization. In this section, we present an example of a verified circular queue implemented using RAR. Circular queues can be found in both hardware and software realizations, making it an ideal example. First, we declare the basic queue structure, as shown below. The maximum queue size can be changed by modifying the `CQ_SZ` constant; ACL2 can reason about arrays of any size.

```
#[derive(Copy, Clone)]

struct CQ {
  front: usize,
  rear: usize,
  arr: [i64; CQ_SZ],
}
```

A typical circular queue operator is the head-of-queue accessor:

```
fn CQ_hd(CObj: CQ) -> (u8, i64) {
  if (CQ_isEmpty(CObj)) {
    return (CQ_EMPTY, 0);
  } else {
    return (CQ_OK, CObj.arr[CObj.front]);
  }
}
```

The enqueue function is as follows:

```
fn CQ_enqueue(value: i64, mut CObj: CQ) -> (u8, CQ) {
  if (CQ_isFull(CObj)) {
    return (CQ_FULL, CObj);
  } else {
   if (CObj.front == CQ_SZ) {  // Insert First Element
     CObj.front = 0;
     CObj.rear = 0;
    } else if (CObj.rear == CQ_MAX_NODE) {
     CObj.rear = 0;
    } else {
     CObj.rear += 1;
    }
    CObj.arr[CObj.rear] = value;
    return (CQ_OK, CObj);
  }
}
```

The circular queue source comprises some 300 lines of RAR code. We use `Plexi` to transpile the RAR source to RAC (not shown), then use the RAC tools to convert the RAC source to ACL2. An example of the translation to ACL2 is shown below for the `CQ_hd` function:

```
(DEFUN CQ_HD (COBJ)
    (IF1 (CQ_ISEMPTY COBJ)
         (MV (BITS 254 7 0) (BITS 0 63 0))
         (MV (BITS 0 7 0)
             (AG (AG 'FRONT COBJ) (AG 'ARR COBJ)))))
```

In this automatically translated function, `AG` is an ACL2 record "get" operation, `MV` provides multi-value return, and `BITS` provides a bit-width specification for a given value.

Similarly, the enqueue function is automatically translated to ACL2 as follows, where `AS` is an ACL2 record "set" operation:

```
(DEFUN CQ_ENQUEUE (VALUE COBJ)
    (IF1 (CQ_ISFULL COBJ)
         (MV (BITS 255 7 0) COBJ)
         (LET ((COBJ (IF1 (LOG= (AG 'FRONT COBJ) 8191)
                          (LET ((COBJ (AS 'FRONT 0 COBJ)))
                                  (AS 'REAR 0 COBJ))
                          (IF1 (LOG= (AG 'REAR COBJ) 8190)
                                  (AS 'REAR 0 COBJ)
                                  (AS 'REAR
                                    (+ (AG 'REAR COBJ) 1)
                                      COBJ)))))
              (MV (BITS 0 7 0)
                  (AS 'ARR
                      (AS (AG 'REAR COBJ)
                          VALUE (AG 'ARR COBJ))
                      COBJ)))))
```

At this point, we can prove theorems about the data structure implementation. We first define a well-formedness predicate cqp for the queue in ACL2. We can then prove functional correctness theorems for the circular queue operations, of the sort stated below:

```
(defthm dequeue-of-enqueue-from-empty
  (implies
   (and
    (cqp CObj)
    (= 1 (CQ_isempty CObj)))
   (= (nth 1 (CQ_dequeue (nth 1 (CQ_enqueue v CObj))))
      v)))
```

ACL2 proves the 35 correctness lemmas and theorems that we have formulated for the circular queue example automatically.

5.2 Crypto Primitives

As noted previously, cryptographic methods are commonly used to enforce zero trust architecture tenets. We have thus ported a majority of the Monocypher cryptography suite [32] (approximately 2300 lines of source code) to RAR. Monocypher is a simple, yet performant and well-maintained set of modern crypto primitives implemented in C. This porting effort was accomplished in two phases: first, the Monocypher C sources were modified to conform to the RAC subset; then that code was ported to Rust/RAR. The initial goal of these first modifications was to ensure that the Monocypher sources were amenable to the use of fixed-size arrays; and if so, to see if there was any appreciable negative performance impact. As it happened, for the selection of crypto primitives that we chose (a cross-section of Monocypher capabilities, from hashing to Encryption/Decryption to Elliptic Curve-based public key functions), the fixed-size array modifications were not that difficult, and as one can observe from columns two and three of Table 1, perfomance was nearly identical after these changes were made. (All results were obtained on one core of a 2020 MacBook Pro with a 2 GHz Intel i5 CPU, 32 GB of RAM, and running MacOS Monterey version 12.0.1.) Importantly, the last column of Table 1 reveals the translation to Rust does not negatively impact execution speed, compared to the baseline.

Function	*Baseline*	*FixedArr/ PassByRef*	*FixedArr/ PassByVal*	*Rust*
Chacha20	423	437	360	389
Poly1305	1157	992	1054	1213
AuthEncrypt	309	328	264	322
Blake2b	636	631	487	735
X25519	9259	10638	7874	9433

Table 1: Monocypher performance comparisons. Higher numbers are better. X25519 results are exchanges/sec; all other results are MB/sec.

One additional performance measure we can readily make is to compare the results of the Monocypher speed tests under pass-by-reference vs. pass-by-value (made possible due to some C preprocessor cleverness). As one can see by comparing columns three and four of Table 1, pass by value reduces C execution speed by 20-30% for most tests.

We were able to translate the Monocypher primitives from RAR to RAC (and thence to both hardware and software synthesis), as well as to ACL2, using the toolchain of Figure 1. Test vectors from the Monocypher regression suite were then used to validate the translation to ACL2, but no significant proof efforts have been undertaken on the translated functions thus far, beyond the necessary loop termination proofs. Future verification plans are discussed in the sections that follow.

6 RELATED WORK

Our work is inspired by, and builds upon, the pioneering work of Russinoff's team at Arm on Restricted Algorithmic C for floating-point hardware verification at scale [29]. Floating-point hardware verification utilizing theorem proving technology has a notable history (*e.g.* [13], [15], [29]). Many of these efforts have focused on engineering artifacts expressed using traditional Hardware Description Languages, such as Verilog; Russinoff's work using an HLS is a notable exception.

A number of domain-specific languages targeting both hardware and software realization have been created. Cryptol [4], for example, has been employed as a "golden spec" for the evaluation of cryptographic implementations, in which automated tools perform equivalence checking between the Cryptol spec for a given algorithm, and the VHDL implementation.

EverCrypt [27] provides a comprehensive verified implementation of modern cryptographic algorithms written in F*, then transpiled to lower-level languages, eventually producing C and/or assembly. We successfully used EverCrypt for our Remote Attestation work on CASE. We initially wished to tie in to the EverCrypt toolchain for our current work, but the lower-level forms produced by the EverCrypt transpilers did not allow us to produce solely fixed-size arrays that we needed for hardware generation. In future, we hope to modify this transpiler machinery to allow us to generate RAC or RAR code, thus allowing us to leverage the significant formal verification work produced by the EverCrypt team.

Rod Chapman recently translated the TweetNaCl compact cryptographic source code suite to the SPARK Ada subset [5], motivated

by a similar desire to ours to produce a cryptographic suite written in a higher-assurance language subset with proof support. Chapman did not, however, contemplate possible hardware implementation. We considered the TweetNaCl sources as a starting point for our work, but Monocypher exhibits superior performance, provides regression and performance testing, and is better written.

Formal verification systems for Rust include Creusot [7], based on WhyML; Prusti [3], based on the Viper verification toolchain; and RustHorn [19], based on constrained Horn clauses. And recently, AWS has announced a model-checker for Rust, Kani [2]. It will be interesting to attempt the sorts of correctness proofs achievable on our system using these verification tools.

7 CONCLUSION

We have developed a prototype environment to enable the Rust programming language to be used as a hardware/software co-design and co-assurance language for critical systems, focusing on systems that implement Zero Trust Architecture tenets. We have demonstrated the ability to establish the correctness of several practical data structures commonly employed in high-assurance systems through automated formal verification, enabled by automated source-to-source translation from Rust to RAC to the ACL2 theorem prover. We have also successfully applied our toolchain to cryptography and data format filtering examples typical of the algorithms and data structures employed in zero trust architecture applications.

We presented two case studies in the development, verification, and validation of zero trust primitives. For the case of an array-based circular queue, we presented the results of full functional verification after automated translation from Rust to ACL2. This was supplemented by test vectors executed using the ACL2 read/eval/print loop, thus providing validation of the translation process. For the case of cryptographic primitives, we detailed how we ported the Monocypher suite first to Russinoff's RAC, and then to the RAR Rust subset. We demonstrated that translation to a fixed-array-size formulation, needed for RAC, had no negative impacts on performance. We then exercised the RAR toolchain on a significant subset of the Monocypher suite, demonstrating the feasibility of expressing cryptographic primitives under the datatype and iterative form restrictions necessary to achieve both hardware and software synthesis, as well as automated translation to the ACL2 theorem prover.

In future work, we will continue to develop the RAR toolchain, increasing the number of Rust features supported by the RAR subset, as well as continuing to improve the ACL2 verification libraries in order to increase the ability to discharge RAR correctness proofs automatically. We will also continue to work with our colleagues at Kansas State University on direct synthesis from architectural models. Finally, we will pursue a connection to the EverCrypt work, as described earlier.

8 ACKNOWLEDGMENTS

This work was funded in part by DARPA contract HR00111890001. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Many thanks to David Russinoff of Arm for answering questions about the RAC toolchain; to Loup Vaillant for developing the excellent Monocypher crypto package; to Geoffry Song for the plex tool; and to John Hatcliff and Robby at Kansas State University for their ongoing pioneering efforts in the area of high-assurance synthesis from architectural models. Thanks also go to the anonymous reviewers for their insightful comments.

REFERENCES

[1] Ron Amadeo. 2021. Google is now writing low-level Android code in Rust. https://arstechnica.com/gadgets/2021/04/google-is-now-writing-low-level-android-code-in-rust/

[2] Amazon Web Services 2022. *Announcing the Kani Rust Verifier Project.* Amazon Web Services. https://model-checking.github.io/kani-verifier-blog/2022/05/04/announcing-the-kani-rust-verifier-project.html?fbclid=IwAR2M_B1IEBfkVhIXSuuAxt3McC_QpUnTuzDq9jG40HOaJzxw8z1Nw9XU_i4

[3] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. 2022. The Prusti Project: Formal Verification for Rust (invited). In *NASA Formal Methods (14th International Symposium)*. Springer, 88–108. https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[4] Sally Browning and Philip Weaver. 2010. Designing Tunable, Verifiable Cryptographic Hardware Using Cryptol. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer, 89–143.

[5] Rod Chapman. 2022. SPARKNaCl: A Verified, Fast Re-implementation of TweetNaCl. In *Proceedings of FOSDEM'22.* https://fosdem.org/2022/schedule/event/ada_sparknacl/

[6] Darren Cofer, Isaac Amundson, Junaid Babar, David Hardin, Konrad Slind, Perry Alexander, John Hatcliff, Robby, Gerwin Klein, Corey Lewis, Eric Mercer, and John Shackleton. 2022 (to appear). Cyberassured Systems Engineering at Scale. In *IEEE Security & Privacy.* https://doi.org/10.1109/MSEC.2022.3151733

[7] Xavier Denis. 2022. *Creusot.* https://github.com/xldenis/creusot

[8] Peter H. Feiler and David P. Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language* (1st ed.). Addison-Wesley Professional.

[9] David S. Hardin. 2020. Put Me on the RAC. In *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-20).* 142–145.

[10] David S. Hardin. 2020. Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems. In *Proceedings of the 2020 IEEE Systems Conference.* https://doi.org/10.1109/SysCon47679.2020.9381831

[11] David S. Hardin. 2022. Hardware/Software Co-Assurance using the Rust Programming Language and ACL2. In *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-22).*

[12] David S. Hardin and Konrad L. Slind. 2021. Formal Synthesis of Filter Components for Use in Security-Enhancing Architectural Transformations. In *Proceedings of the Seventh Workshop on Language-Theoretic Security, 42nd IEEE Symposium and Workshops on Security and Privacy (LangSec 2021).* https://doi.org/10.1109/SPW53761.2021.00024

[13] John Harrison. 2006. Floating-Point Verification Using Theorem Proving. In *Formal Methods for Hardware Verification*, Marco Bernardo and Alessandro Cimatti (Eds.). Springer Berlin Heidelberg, 211–242. https://doi.org/10.1007/11757283_8

[14] John Hatcliff, Jason Belt, Robby, and Todd Carpenter. 2021. HAMR: An AADL Multi-Platform Code Generation Toolset. In *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA) (LNCS, Vol. 13036).* 274–295.

[15] Warren A. Hunt, Sol Swords, Jared Davis, and Anna Slobodova. 2010. Use of Formal Verification at Centaur Technology. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*, David S. Hardin (Ed.). Springer, 65–88.

[16] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press.

[17] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[18] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[19] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 15 (oct 2021), 54 pages. https://doi.org/10.1145/3462205

[20] Mentor Graphics Corporation 2016. *Algorithmic C (AC) Datatypes.* Mentor Graphics Corporation. https://www.mentor.com/hls-lp/downloads/ac-datatypes
[21] Mentor Graphics Corporation 2020. *Catapult High-Level Synthesis.* Mentor Graphics Corporation. https://www.mentor.com/hls-lp/catapult-high-level-synthesis/
[22] Eric Mercer, Konrad Slind, Isaac Amundson, Darren Cofer, Junaid Babar, and David Hardin. 2021. Synthesizing Verified Components for Cyber Assured Systems Engineering. In *24th International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021).* https://doi.org/10.1109/MODELS50736.2021.00029
[23] Matt Miller. 2019. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
[24] Shane Miller and Carl Lerche. 2022. Sustainability with Rust. https://aws.amazon.com/blogs/opensource/sustainability-with-rust/
[25] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673
[26] Adam Petz, Grant Jurgensen, and Perry Alexander. 2021. Design and Formal Verification of a Copland-based Attestation Protocol. In *ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE 2021).*
[27] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. 2020. EverCrypt: A Fast, VeriïňĄed, Cross-Platform Cryptographic Provider. In *IEEE Symposium on Security and Privacy.* IEEE. https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-veri%ef%ac%81ed-cross-platform-cryptographic-provider/
[28] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. 2020. *NIST Special Publication 800-207: Zero Trust Architecture.* National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-207
[29] David M. Russinoff. 2022. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach* (second ed.). Springer. https://doi.org/10.1007/978-3-030-87181-9
[30] Geoffry Song. 2020. *plex: a parser and lexer generator as a Rust procedural macro.* https://github.com/goffrie/plex
[31] Jeff Vander Stoep and Stephen Hines. 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html
[32] Loup Vaillant. 2022. *Monocypher: Boring Crypto that Simply Works.* https://monocypher.org
[33] Xilinx, Inc. 2018. *Vivado Design Suite User Guide: High-Level Synthesis.* Xilinx, Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf