Survey of Approaches for Security Verification of Hardware/Software Systems

Onur Demir[1], Wenjie Xiong[2], Faisal Zaghloul[2], Jakub Szefer[2]

[1] Yeditepe Üniversitesi, Istanbul, Turkey
odemir@cse.yeditepe.edu.tr
[2] Yale University, New Haven, CT, USA
{wenjie.xiong, faisal.zaghloul, jakub.szefer}@yale.edu

Abstract

Variety of computing systems have been proposed to provide protection for sensitive code or data through hardware or software mechanisms. This paper surveys the landscape of security verification approaches and techniques for hardware/-software systems at different levels: from a software-application level all the way to the physical hardware level. Different existing projects are compared, based on the tools used and security aspects being examined. Since many systems require both hardware and software components to work together to provide the system's promised security protections, it is no longer sufficient to verify the software levels or the hardware levels in a mutually exclusive fashion. This survey highlights common sets of system levels that are verified by the different existing projects and presents to the readers the state of the art in hardware security verification. Few approaches come close to providing full-system verification, and there is still much room for improvement. In this survey, readers will gain insights into existing approaches in formal modeling and security verification of hardware/software systems, and gain insights for future research directions.

1 Introduction

News articles and opinion pieces by top security researchers constantly remind us that as computing becomes more pervasive, security vulnerabilities are more likely to translate into real-world disasters [124]. In order to promote information security, many hardware/software mechanisms have been designed. However, if the designs are not perfect, they are vulnerable to attackers, resulting in the failure of protection. Formal methods provide the possibility of ruling out every security vulnerability in the designed system. Formal methods have been used in the functional verification of hardware and software for a long time. Recently, the use of formal methods for security verification of both the hardware and the software of a system has emerged as an important research topic.

Since the security of the systems depends on the correctness of the protections that both the hardware and software components provide, there is the need to verify the security of both the software and, especially, the hardware components

at design time. Unlike software, hardware is almost impossible to patch once it is fabricated. The lack of formal verification may leave some vulnerabilities in the designed system. For example, Intel's Core 2 Duo processor family is known to have 129 bugs [68]. The number of known bugs illustrates the limitations of the existing verification approaches; and these do not even consider security yet. This has motivated researchers to look into the verification of the security properties of these systems at design time, and into developing new methods for the security verification of hardware/software systems, as outlined in this survey.

There is a lack of familiarity with verification tools among researchers and academics working on hardware/software security architectures. This results in designs that are only tested with certain security property test cases, which may not be exhaustive, and architectures whose security properties lack formal specifications. Many hardware/software "secure architectures" have also been designed which provide enhanced security features in hardware, but most, if not all, of these designs do not come with a formal proof of security properties. In academia, many secure architectures have been designed [91, 135, 74, 83, 43, 26, 70, 136, 145]. The absence of formal verification of the security of system architectures may be one reason why industry has been very slow to adopt the academic research in this field. Nonetheless, a number of designs from processor vendors also provide some hardware features for security, e.g. ARM TrustZone [140], Intel SGX [100], and most recently AMD Memory Encryption [4]. These designs all rely on the implicit assumption that the hardware is correct. The industry designs we know about do not have any publicly available formal security guarantees nor specifications.

In this survey, we compare projects that consider both hardware and software and use formal methods to verify the security of computer systems. We show the state of the art in security verification of hardware/software and lower the barrier to entry into this field for interested researchers.

1.1 Software and Hardware System Levels Considered in Verification Process

A system is typically composed of multiple hardware and software levels, as shown in Figure 1. The typical software levels in a computer system are: Application, Operating System, and Hypervisor. These levels represent typical software levels for a commodity processor. The typical hardware levels in a computer system, as shown in Figure 1 are: ISA (Instruction Set Architecture), Microarchitecture, RTL (Register Transfer Level), Gate, and Physical.

Traditionally, upper levels depend on the lower levels for functionality and security. A guest OS relies on the hypervisor to provide isolation from other malicious guests; a secure operating system cannot protect information leakage if the underlying hardware has a backdoor. Meanwhile, an ISA requires a microarchitecture that implements it, a microarchitecture requires an RTL that implements it, and so forth. The relationship is not strictly linear in that upper level always depends on all lower levels. The verification process needs to consider which levels are important for system verification. Some of the secure

hardware/software systems have introduced hardware or instructions that allow higher levels to be protected from intermediate levels. For example, in Bastion [26]applications are able to communicate with the hypervisor bypassing the OS, while in HyperWall [136] a virtual machine can bypass the hypervisor, and the hardware provides some of the basic memory management functionality.

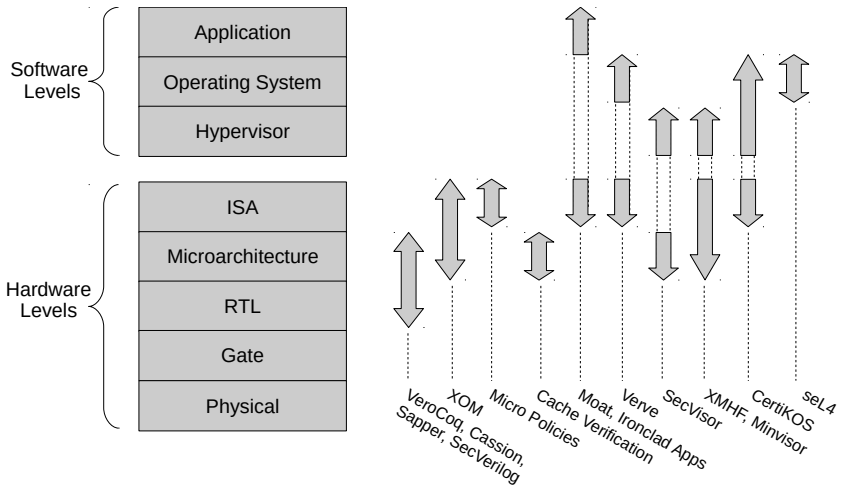

Fig. 1. Hardware and software levels found in a typical commodity computing system. The verification projects surveyed in this work focus typically on one or more levels. Broadly, projects either focus on spanning multiple software levels, or hardware levels; some projects can span both software and hardware levels. Tables 2 and 3 at the end of the survey present summary of the projects and the levels they cover.

The software and hardware levels needed for correct operation constitute the trusted computing base (TCB) which contains all the software and hardware that need to be trusted (although may not be trustworthy). The TCB should be verified for security to make it truly trustworthy. A number of software security verification work has focused on the software levels only, assuming the hardware is trusted and correct [47]. Only more recently, researchers started looking at security verification that considers both hardware and software levels, as outlined in this survey. Effectively, TCB consists of some components in some levels, and verification tools and methods should include all the levels in TCB. Figure 1 on the right side shows the various surveyed projects and the different system levels that their security verification covers. The details of these projects are given in Sections 4 and 5, which highlight the different system levels each project considers for security verification.

1.2 Survey Outline

The remainder of the survey is structured as follows. Section 2 discusses various security properties that designers want to prove about their systems. Section 3 discusses various tools used for security verification. Section 4 presents survey of hardware focused works. Section 5 presents survey of software focused works. Section 6 summarizes our findings. Section 7 concludes the survey.

2 Overview of Security Properties

Security verification of a system makes sense in a context where the system explicitly claims to provide some security guarantees, and where there exists a Trusted Computing Base (TCB) upon which those security guarantees depend. The TCB is comprised of one or more layers previously shown in Figure 1. As these layers work together to implement the system operation, they need to be verified. The key security properties are discussed below.

2.1 Confidentiality, Integrity, and Availability

There are typically three major security properties that we consider when analyzing a system: confidentiality, integrity, and availability [82].

- "Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities."
- "Integrity is the prevention of unauthorized modification of protected information without detection."
- "Availability is the provision of services and systems to legitimate users when requested or needed."

From a security perspective, confidentiality, integrity, and availability assume a sophisticated attacker who attempts to maximize their chance of breaking the system. Security verification assumes that reliability, i.e. protection from random faults or errors, is already provided by the system, and focuses instead on the deliberate attacks by a smart adversary.

Of the three properties, confidentiality and integrity are provided more commonly. Availability is not often considered in the surveyed papers, as it is difficult to provide and prove; an attacker can simply turn off the power, disable network connections, or use a plethora of other methods make the system unreachable. If a system is not available, it does not disclose any information to the attacker, nor does it allow information to be modified; the system is just unusable. Thus, availability is rarely proved as a part of security verification for a single system – availability can be achieved through a mixture replication, distribution, and power failover.

2.2 Non-interference

Non-interference is a property that typically refers to how *low*-security entities interact with *high*-security entities. The interference between these entities is analyzed to confirm that the low-security entity should not be able to observe any differences in its own behavior, or the behavior of the system, in response to, or as a byproduct of, a high-security entity processing sensitive or non-sensitive inputs. The high-security entity, however, may observe differences in the behavior of system or the low-security entity. Non-interference essentially means that information about the operation of the "secure" entity does not leak to the "insecure" entity. Non-interference can be used to verify confidentiality, when a system is partitioned into isolated entities such as low/high, secret/public etc.

2.3 Isolation

The isolation property is closely related to non-interference, and typically deals with the separation of two entities which are considered equal (i.e. the isolation of two virtual machines from each other). Through isolation, one component should not be able to affect integrity or confidentiality of another component. Isolation is often considered as part of functional verification, but in such scenarios only data isolation is considered. The timing information observations are typically added if isolation is to be proved as a part of security verification.

2.4 Information Flow

Information flow is actually both a security property and also a technique used in verification. The information flow property refers to the transfer of information between different entities. Information flow can be explicit, e.g. $a = b$ where data or information in b is moved to a; or it can be implicit, e.g. $b = 0;\ if(a)\ then\ b = 1$; where the value of b reflects whether a is true, but there was never a direct assignment, or copying of data, from a to b. Again, typically when discussing information flow there is a *low*-security entity that interacts with a *high*-security entity. A system could have a desired property such as "there is no information flow between the components x and y" or "component x's file z is never accessible by component y". Information flow can happen through data or through timing information.

2.5 Type and Memory Safety

Type and memory safety are software-centric properties. Type safety is a programming language level property which ensures that data may only be copied between variables of same type. Meanwhile, memory safety deals with how memory is accessed, and prevents errors such as buffer overflow. Type and memory safety focus on correctness of the program or system itself, by proving these properties it can be ensured that bugs cannot be easily exploited in the program, since bugs could lead to confidentiality or integrity breaches.

2.6 Memory Integrity, and Execution and Code Integrity

Note that memory integrity and memory safety are two different concepts. As explained above, memory safety is a software development concept. However, memory integrity requires hardware mechanisms as well. As explained above, integrity prevents unauthorized modification of protected data. Memory integrity treats the whole memory or a part of memory such as a set of virtual machines, processes, or parts thereof, as protected data. The unauthorized modification of these areas is prohibited. Execution and code integrity is a special subset of memory integrity where the code segment, of a process or a virtual machine, and the data associated with it are treated as protected.

3 General Tools and Mechanisms

This section presents a background on the different tools, mechanisms, and approaches typically needed to check security guarantees, which we discussed in the previous section. The general flow of the security verification process is shown at a high level in Figure 2. The starting point is the actual system, either an already existing system or a design of some new system whose security properties need to be verified. From the actual system, or design, a representation of the system needs to be obtained in the verification tools, (a) in Figure 2. In parallel, the security properties of the system need to be specified, (b) in Figure 2. The security properties are closely tied to the system's assumed threat model. The security properties can be specified separately or together within the representation of the system, in which case (a) and (b) would be done together. The final step is the actual verification process which takes the system representation and security properties as input, and returns whether the verification passed or failed, (c) in Figure 2. If the verification fails, the design needs to be updated and re-evaluated, (d) in Figure 2.

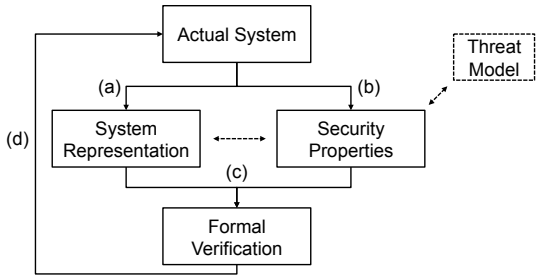

Fig. 2. General procedure for security verification.

3.1 System Representation

In order to check if a system complies with some properties, we need a representation of the system that expresses the behavior of the system accurately. Ideally, the actual system description can be used, such as the source hardware description language (HDL) code for hardware components, or a programming language source code for software components. Otherwise, a model in the verification tool is needed. One reason a model may be needed is that the way a system is described in HDL or programming language may not be compatible with the verification tool that is being used, or the way the system is described is too complex for the verification process to handle.

Hardware components can be described with Hardware Description Languages (HDLs). The most popular HDLs are Verilog [138] and VHDL [93]. Recently, a new tier of HDLs is emerging that feature more high-level abstractions and reusability than circuit-level HDLs. This new generation of HDLs is called Hardware Generation Languages (HGLs) [95], including Chisel [7], BluSpec [109], and Genesis2 [128]. There are some attempts to link hardware and circuit system representation methods with security-property specification towards security verification, which we will present in Section 4.

Software components can be described by their high-level implementation in programming languages such as C, C++, or Java. There are also ongoing efforts to integrate design processes and system specification with security-property specification and including verification information inside programming languages as annotations. Examples include TAL [33] and Dafny [85].

Alternatively, some tools require a model of the system based on its original description. For example, this survey later discusses VeriCoq which is a tool that can be used to translate (annotated) Verilog code to code understood by Coq verification tools. If automated method of creating a model is not available, then model has to be created manually by engineers. However, when creating models manually, proving the correspondence between the model and the actual system is an open research problem.

Formal verification is done with respect to a system representation, as described above. Most projects assume a trusted compiler or tool chain such that the system realization indeed matches the system representation, and does not contain extra hidden, or unwanted, functionality that may compromise the security of the system. For example, after verifying the C code of an application, there is still a concern that the compiler may not generate the correct machine code from the C code. A malevolent compiler might insert malicious code into the binary, as demonstrated in [139], where a virus-infected compiler as able to inject back-doors into applications during compilation. A number of projects include "trusted" compilers that are guaranteed not to inject behavior that was not specified. One example of such a compiler is CompCert [87] which is a certified compiler that generates binaries from Coq code. All surveyed work assumes trusted toolchains.

3.2 Security Properties

In Section 2, we listed the security properties that designers may be interested in checking. Depending on the verification mechanism, either *deductive mechanism* or *algorithmic mechanism* in Section 3.3 and 3.4, the security properties are represented in the corresponding verification tools. In *deductive mechanisms*, the security properties can be represented in terms of logical formula. A logical formula serves as a constraint on the states the system is allowed throughout its execution. Some specialized forms of logic are used to express the relations between the states of the system. In *algorithmic mechanisms* security properties can be expressed as invariants within a system, and their validity is checked against all possible execution paths.

3.3 Formal Verification via Deductive Mechanisms

When using deductive mechanisms, verification is achieved by deducing properties from a system representation. Theorem provers fall in this category. The key element in deductive mechanisms is a proof. Deductive mechanisms use formal proofs to verify that a system complies with some given properties. Security verification can be done using theorem provers in a number of ways as outlined in [112]:

- General Property Verification – For security verification by theorem proving, a security property can be represented in an theorem prover by a type $\mathcal{T}$, which is similar to types in programming languages. Type $\mathcal{T}$ represents all the systems that have the given specific security property. For example, a type can be used to represent the non-interference property. If system $\mathcal{P}$ can be proved to be of type $\mathcal{T}$, then the system is verified. Otherwise, it means that the system does not comply with the given properties of type $\mathcal{T}$ and needs to be redesigned. In this approach, any system that can be represented as $\mathcal{P}$ can undergo same verification.
- Individual Verification – In a different approach, the system $\mathcal{P}$ can be treated individually. In this case, there is no need to define security properties as types. One can express the desired property directly, e.g. *"the program $\mathcal{P}$ complies with the security property $\mathcal{A}$"*, as mathematical statements in a theorem prover. If this statement is proved to be true in the theorem prover, then the system passes the verification. Otherwise, the system is redesigned accordingly. There is a lot of effort needed for this type of proofs, and the method is not general.

The above approaches can be realized with the help of a number of theorem provers. Theorem provers (also referred to as proof assistants) aid the verification process by providing frameworks for creating a mathematical model of the system, for specifying the security properties, and for formally proving whether the model complies with the properties or not. Theorem provers are generally composed of a language (such as Coq), and an environment for describing the

proofs (such as CoqIDE). There is a number of proof assistants used actively in academia and industry such as: Coq [14], Isabelle/HOL [110], PVS (Prototype Verification System) [111], ACL2 (A Computational Logic for Applicative Common Lisp) [71], and Twelf (LF) [55]. Theorem proving typically requires a lot of effort and time to complete, and learning the required tools is seen as one of the difficult aspects of verification using theorem provers. In the following paragraphs we will introduce different theorem provers and give examples on their usage in functional and security verification.

Coq [14] is based on higher-order typed lambda calculus which is referred to as Calculus of Constructions. Coq is composed of two parts: the Coq language and the environment. In the first part, there is a Coq language in which the primitive functions, definitions, types, and sorts (types of types) are defined. Coq language is also used to define new objects, functions, propositions, and proofs. It is implemented in ML which is a functional programming language [103]. The second part contains an environment for designing theorems and proofs (CoqIDE or Emac's Proof General) with some support for extensible notations, tactics for proof automation, and support libraries. Coq's logical deduction system follows Curry-Howard isomorphism which is a direct relationship between programming and mathematical proofs.

Isabelle/HOL [110] is a theorem prover based on higher-order logic. It is a branch of Isabelle [113] which is a more generic theorem prover that is also used to express logical formalisms. Isabelle/HOL has been used to verify a number of systems. Most noteably, it was used for functionally verifying the seL4 [75] operating system, the Java compiler [133], verifying high level C applications [122] and many others [20, 6, 84, 37, 38]. Isabelle/HOL is also implemented in ML.

ACL2 (A Computational Logic for Applicative Common Lisp) [71] is theorem prover based on first-order logic. It consists of a language and a theorem prover. ACL2 language is a variation of Common LISP [72]. There are many examples of ACL2's usage in industry. In [52] the authors used ACL2 for the security verification of a security policy within the Rockwell Collins AAMP7 separation kernel. Other examples fall in functional verification category as in [107], [119], [118].

Twelf is a proof assistant which uses LF (The Edinburgh Logical Framework) logic [55]. LF is based on a general treatment of syntax, rules, and proofs by means of a typed-calculus with dependent types. Twelf consists of a logic programming interpreter, a reasoning environment, and a module system [114], [125]. Several research projects have been using Twelf for verification. In [33], Twelf was used to verify the soundness of typed assembly language. In [81], it is used to verify Standard ML.

Other interactive theorem provers exist which we could not all list above. [144] describes 17 interactive theorem provers and demonstrate how a theorem can be proved in each tool. The interactive theorem provers listed in their paper are HOL [49], Mizar [50], PVS [111], Coq [67], Otter/Ivy [99], Isabelle/Isar [143],

Alfa/Agda [54], ACL2 [71], PhoX [137], IMPS [45], Metamath [102], Theorema [23], Lego [96], Nuprl [76], Ωmega [11], B-method [1], and Minlog [126].

Pen-and-Paper Proofs Although theorem provers are mostly used for proving security properties, manually proving security properties with pen and paper is still a method used by developers. Some projects such as Sapper [88], Caisson [89], and SecVerilog [148] in Section 4 use pen-and-paper proofs. In these projects, a new language is built. The system representation and security property specifications are written in the new language, and translated into Verilog. The designers proved on paper that the translation process guarantees a selected security property. In this case, any future system designers who use the tool do not need to prove anything as long as the translator is not modified. Prior works show it is feasible to verify on paper the part of the system that are unlikely to change, e.g., the translators in Sapper, Caisson and SecVerilog.

3.4 Formal Verification via Algorithmic Mechanisms

Algorithmic mechanisms typically use an algorithmic search, which is performed over a system's representation and its states, rather than using deduction. Model checkers (Section 3.4), SMT (Satisfiability Modulo Theories) Solvers (Section 3.4), and Symbolic Execution (Section 3.4) fall in this category.

Model Checking According to [8] "model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for a given state in that model." The security property that is being verified has to be defined using a logical form. After the model and the property definition, the model checker can be run to see if the given security property is valid in the system model. The checks can be done either for each transition or each state using invariants, pre- and post-conditions. The execution time of the model checker depends on the complexity of the model. The output can be positive (property satisfied), negative (property violated), or the execution runs indefinitely. Model checking has a well-known state explosion problem, which is the exponential growth rates of states. This may lead to memory insufficiency or extremely long run times. For fairly complex systems, model checking needs to use more abstraction to reduce the number of the states. However, as the level of abstraction gets higher, we run the risk of missing some important details of the system design.

Modeling should represent the behavior of the system. Finite state reachability graphs are the most common way of modeling applications. Reachability graphs or Kripke Structures are designed as a graph where the nodes represent each reachable state while the edges represent the transitions between states [34]. The properties of the system are modeled using a finite set AP of atomic propositions. A Kripke Structure can be defined as $K = (S, S_0, R, L)$ where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is transition relation between states, and L is labeling function that labels a state with atomic propositions $L : S \to 2^{AP}$.

To formally express the behavior of a system over time with respect to certain properties, we use *temporal logic*. There are several varieties of temporal logic:

- *Linear Temporal Logic (LTL)* is a common type of temporal logic where execution paths can be checked [73]. LTL is formulated by a finite set of propositions P, boolean operators *not* ($\neg$), *and* ($\wedge$), and basic temporal operators $Next$, $Previous$, $Until$, and $Since$ (N, R, U, S). A temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$,where each state s_j gives a value for the variables in p. Given a model σ, a temporal formula p holding at a position j in σ is denoted by $(\sigma, j) \models p$. If $(\sigma, 0) \models p$, it is described as p holds on σ, and denoted it by $\sigma \models p$. A formula p is called satisfiable if it holds on some model. A formula is called temporally valid if it holds on all models.
- *Computation Tree Logic (CTL)* shares its basic operations with mathematical logic ($\neg, \wedge, \vee, \Rightarrow, \iff$). However, it also supports operations to check path operators. These operators indicate if a condition is met within one, some, or none of the paths in an execution path [47]. Universal ($\forall$) and existential ($\exists$) operators are used to determine if a condition is met in one or more execution paths in the whole reachability graph. The unary operator $\diamond\psi$ (eventually) is used to indicate if a path is going to satisfy a property at a given time. The unary operator $\circ\psi$ (next) is used to indicate if a property is satisfied in the next state of a path. The unary operator $\Box\psi$ (always) is used to indicate if a property is satisfied in all states of a graph. The operator $\psi\mathcal{U}\phi$ (until) is used to check if a property is satisfied in a path starting at ψ until the path meets ϕ.

Note that there are other variations of temporal logic which do not present here. In the following paragraphs, we introduce some of the important model checkers, and illustrate their use in formal verification. For further details about model-checker design, we point the readers to an early survey by Clarke et al. [30].

SPIN [63] is a model checking tool that supports verification of software models with respect to properties expressed in LTL. SPIN is designed to for the verification of multi-threaded software systems [132]. SPIN includes a specification language called PROMELA [62] which allows modeling distributed systems. SPIN has been used for functional verification of protocols [106, 117, 80], industrial control software [44, 65], and mission-critical software [64, 58, 57, 48, 123].

Murφ [42] is a model checking tool that provides a compiler and a description language where a finite-state concurrent system model and properties can be defined. The finite-state concurrent system is modeled after the actual system being verified. After defining the system, Murφ model checker generates all reachable states using breadth-first search algorithm. Then Murφ checks the given properties of the system in all reachable states. If the model checker reaches a state where the properties do not hold, it returns with an error. Murφ is used for the functional verification of cache coherence protocols [29], cryptographic protocols [104, 105], network protocols [129], and critical software verification [35].

SMV (Symbolic Model Verifier) is a model checker for checking specifications written in CTL in finite-state systems [101]. SMV is used for the verification of PLC (Programmable Logic Controller) programs [116], communication protocols [66], and security protocols [31].

CBMC (C Bounded Model Checker) [78] is a model checking tool for C and C++ programs. CBMC can verify bounded assertions in programs such as division by zero, array bound check, pointer checks, arithmetic overflow checks and user defined assertion checks. CBMC is widely used for proving safety of software systems such as [10], [18], [79], [22] and [77].

SMT Solvers Satisfiability Modulo Theories (SMT) solvers are used to solve satisfiability problems expressed in first-order logic with respect to some logical theory. For verifying systems, first they have to be transferred into formulas that SMT solvers can work with. The validity of the property is then checked by SMT solvers [13]. Note that SMT solver execution time can vary from a few seconds to hours depending on the size of the problem. There a number of SMT solvers available for public or commercial use. Here, we describe Z3 which is an SMT-based solver, then we discuss an intermediate verification tool, Boogie, and finally we overview a high-level language with verification abilities, Dafny. We will present a number of projects that use these tools in Section 5.

Z3 [40] is an SMT solver from Microsoft Research. It is a low-level tool for building other tools (such as Dafny and Boogie) that require solving logical formulas. Z3 follows SMT-LIB 2.0 standard which defines the standard API for SMT solvers [9]. Z3 can be used as a scripting language as well. Z3 can be used as a verifier if the programs are expressed in first-order logic. The difficulty of such an effort resulted in higher-level tools that reside between the programming language and Z3, such as Boogie, or entire languages that support annotations to help to construct logic formulas, such as Dafny.

Boogie [147] is an intermediate verification tool and language that use Z3 for providing a layer on which verifiers can be built for other languages. For example, there are verifiers that use Boogie for high level languages such as C, Dafny, Chalice, and Spec#. Boogie was used for creating tools for the functional verification of race and divergence-free GPU Kernels [15], and prove type and memory safety of the Verve [146] operating system.

Dafny [85] is a high-level language that uses Boogie for verifying programs during compilation. Dafny supports verification by using high-level annotations to keep track of correctness of the system. Security invariants are assured throughout the code by maintaining properties that has to hold throughout the execution time. These properties are checked using pre and post-conditions within the code. Dafny has been used for high level software and system verification [86, 97, 60], as well as algorithm verification [115].

Symbolic Execution In symbolic execution [25], symbols are used in lieu of the actual values of program variables or hardware components. As the simulation executes, the symbols can be updated, combined, turned into expressions,

and sometimes simplified. Since the values of the symbols may not be clearly identified, each possible execution-branch is followed. In this way, all possible execution states can be evaluated simultaneously, at great cost of storage and slow execution during testing. There are variations of this technique, such as symbolic trajectory evaluation [61]. Unlike model checking, which requires a model of the system, symbolic execution deals directly with the source code, in case of software applications. The source code is executed in an interpreted environment that is able to keep track of all possible values of all variables, pointers, etc., of the code as it runs.

In [134], the authors used symbolic simulation for verifying security properties of firmware by introducing a property specification language and an algorithm based on symbolic execution to verify these properties. Symbolic execution is also used to verify the security properties of cryptographic protocols [2]. There are many symbolic execution tools including S2E [28], TRACER [69], Woodpecker [36], Triton [121], and Java PathFinder [59].

4 Security Verification focusing on the Hardware Levels of a System

In this section, we present projects which focus on the security verification of the hardware levels. There are three groups of approaches to verify the design: using proof assistants, using verified hardware languages with information flow tracking, and using model checkers. In the first group, the hardware design is first converted to an equivalent representation that works with a proof assistant, and then the designers write the proof of the design in the proof assistant as well. The second group comprises of new HDLs that feature information flow tracking annotations. It is proven that the Verilog code generated from the new HDL has some security properties, e.g. confidentiality. So hardware designers can consider security at design time by adding annotations in their code. It is guaranteed that the generated HDL design would have the desired security properties. In the third group, given the description of the architecture or system operation, a finite state model is created. The desired security properties are represented by invariants in the model. The model checker performs a search to check whether the security invariants always hold.

4.1 Hardware Design Security Verification using a Proof Assistant

VeriCoq VeriCoq is a tool that provides mechanisms to transform Verilog code into code with PCHIP (Proof-Carrying Hardware Intellectual Property), which makes it possible for the customers to verify the security of the design written in Verilog. In addition, the newer version of VeriCoq also supports the verification of the information flow property [17].

The verification process is shown in Figure 3. Given the functional specification by the customer, the designers implement the system in Verilog, (1) in

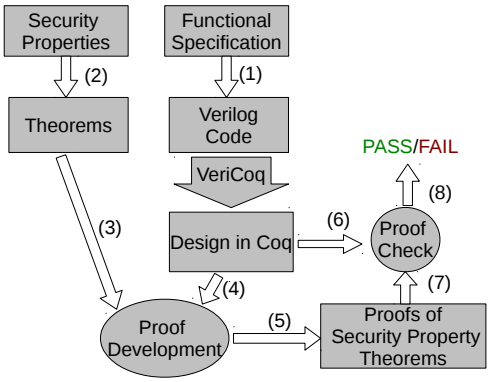

Fig. 3. VeriCoq design process.

Figure 3. Verilog code is then converted into Coq by VeriCoq. Based on the security properties requested by the customer, designers create the theorems to be verified, (2). With the design represented in Coq, alongside with the theorems, developers come up with formal proofs showing that the code has the desired security properties, (3), (4), and (5). The customer receives both the design in Verilog and the proofs. They can then verify that the design aligns with the defined security properties by using Coq, (6) and (7) in Figure 3. CoqIDE will either pass the design, meaning that it conforms the properties, or fail it, (8) in Figure 3.

The basic advantage of VeriCoq is the automated conversion of Verilog code into Coq. VeriCoq supports an essential subset of Verilog. The newer version of VeriCoq addresses information flow verification and automation of PCHIP framework at the same time [16]. The information flow proofs need an extra step for labeling the variables in the design. VeriCoq then implicitly creates theorems and proofs for guaranteeing the information flow property. VeriCoq is still an ongoing project where the developers aim to extend its capabilities to support more Verilog statements.

Micro-Policies A recent work on Programmable Unit for Metadata Processing (PUMP) [41] allows flexible computation of metadata alongside with the main computation with a modest performance overhead. PUMP allows programmers to create policies and rules that enforce IFT mechanisms by manipulating the metadata tags in each instruction. Metadata processing can thus support many safety and security policies, e.g. memory safety, control-flow integrity, confidentiality, etc. However, given a high-level specification, it is nontrivial to design metadata processing rules. Whether the metadata processing rules in PUMP comply a high-level security properties needs to be proved. Micro-Policies represent a generic framework for formalizing and verifying arbitrary policies in PUMP architecture.

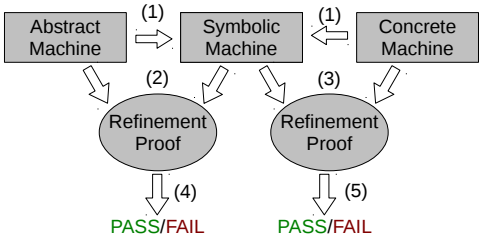

Fig. 4. Micro-Policies verification process.

To design a set of metadata rules, first, an abstract machine specification with a set of instructions and information flow policies is defined, showing what security properties the machine should have. Then, programmers design the metadata rules (concrete machine), where the information flow policy is implemented into the PUMP hardware. To reason whether the concrete machine reflects the abstract machine specification, an intermediate layer *symbolic machine* is added, as shown by arrows labeled (1) in Figure 4. The Micro-Policies prove the equivalence by backward refinement, which means if there is a state transition in low-level machine, there exists a corresponding transition in high-level machine. Using Coq, it can be formally proven whether the concrete machine backward refines the symbolic machine (2), and whether the symbolic machine backwards refines the abstract machine (3). With the backward refinement verification, (4) and (5), the designers make sure that concrete machine has the security properties of abstract machines. The micro-Policies paper shows the proof of a variety of security policies, including sealing, compartmentalization, control flow integrity, and memory safety [39].

A Comparison of VeriCoq and Micro-Policies VeriCoq and Micro-Policies provide us with a better view of how verification can be handled in different domains. Their common character is to use Coq for theorem proving. However, in VeriCoq the development starts with Verilog and in Micro-Policies it starts with Coq. In VeriCoq, the code is directly verified by proving that the security properties are valid for the code. On the other hand, Micro-Policies are designed for systems where directly proving security properties might be very hard. Micro-Policies offer a solution which employs refinement strategy.

4.2 Security Verification Leveraging HDLs with Information Flow Tracking

Information Flow Tracking (IFT) has been widely used to enforce security properties, such as confidentiality, integrity, and non-interference [120]. Programming language techniques can be used to specify and implement information flow tracking policies. While most of the works exploring IFT focus on the software domain, there is no reason it cannot be used in hardware design. Caisson [89],

Sapper [88], and SecVerilog [148] introduce information flow tracking into Verilog.

Information flow tracking guarantees that the security policy is applied to all information flows in a system. The policy, access groups and permission, rules and exceptions are designed to achieve high-level security properties. Assuming correctness of the policy, the systems can defend against information leakage using information flow control. The following projects embed IFT in Verilog, and also prove the information flow policy guarantees non-interference.

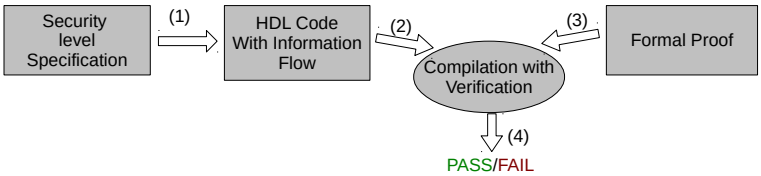

Fig. 5. Static information flow verification process in Caisson and SecVerilog.

Caisson Caisson [89] is a hardware description language with static information flow verification at design time. The verification process is shown in Figure 5. First, data ports in a hardware module are assigned with security levels (i.e. Low, $High$). The security goal of the design is to make sure there is no information flow form $High$ to Low, (1) in Figure 5. The design is written in Caisson with the security labels on each register and wires. In this way, information flow is considered at design time, (2) in Figure 5. During compilation, it is checked whether the information flow strictly follows the defined policy. If so, there will be no information flow form $High$ to Low during the system runtime, which means security is statically-verifiable. The design's Verilog code can be generated from Caisson through the compiler, and can then be synthesized using existing tools, (4) in Figure 5. In the paper, it is formally proved that Caisson type system enforces timing-sensitive non-interference in designed hardware, (3) in Figure 5.

Using Caisson, the authors were able to create the first provably information-flow secure processor that contains a time-multiplexed pipeline and a partitioned cache. The design exhibited only minor overhead. The pipeline is secured by secure time multiplexing (or time lease) with separated context (i.e., registers and memory) for each security level. The cache is statically partitioned between different security levels.

Sapper Sapper [88] is a hardware description language that automatically enforces security policies in hardware logic. It is an extension of a synthesizable subset of Verilog. Figure 6 shows the design flow: (1) A system with a set of input and output ports each of which has a security level is considered, e.g. high (H) and low (L). The hardware logic should ensure that data flowing to

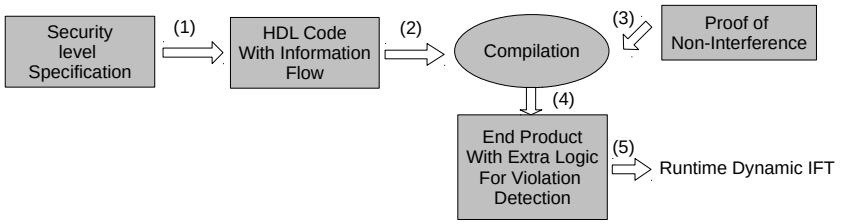

Fig. 6. Dynamic information flow verification process in Sapper.

any output port follows the information flow policy. In the presence of an active attacker (e.g. a malicious software in the system), who has full control over all L input ports, the non-interference enforced by the hardware can protect all the data tagged with H. Non-interference here means if an L-observer can not distinguish between two configurations at the beginning of each clock cycle, then she should not be able to distinguish between them at the beginning of next cycle. (2) Security labels indicating an IFT policy is inserted to the code in Sapper language (highlighted in Figure 7). If the security check shows that the IFT policy is violated, the hardware should not pass the information on. (4) At compile time, Sapper statically analyzes the hardware logic and automatically inserts dynamic IFT logic. Figure 7 shows how Sapper inserts the information flow checking and tracking logic using tags in the generated Verilog code. Static analysis enables the system to cover explicit, implicit, and timing-based information flows. With the inserted IFT logic, the synthesized hardware can track and check security policy at runtime, and any policy violations will be detected. (3) A proof sketch of non-interference of the system generated by Sapper is given in the paper.

With Sapper, a processor is designed and simulated in ModelSim [56]. A micro-kernel and a compiler are also implemented, and processes in different security levels can run on the processor. Unlike Caisson [89], data with different security labels can share resources in Sapper, e.g. registers, resulting in a lower overhead.

	Sapper	Verilog

check	`reg[7:0] a: L;`
	`reg[7:0] b,c;`
	`a <= b & c;`
		`reg[7:0] a,b,c;`
		`reg a_tag,b_tag,c_tag};`
		`if (a_tag >= (b_tag|c_tag))`
		`  a <= b & c;`

track	`reg[7:0] a,b,c;`
	`a <= b & c;`
		`reg[7:0] a,b,c;`
		`reg a_tag,b_tag,c_tag};`
		`a <= b & c;`
		`a_tag <= (b_tag|c_tag)`

Fig. 7. Sapper code and generated Verilog code, modeled after [88]. Security label is highlighted.

SecVerilog SecVerilog [148] is also Verilog extended with information flow annotation. It enables static check of hardware information flow. Unlike Sapper, which uses dynamic information flow at runtime, information flow checking in SecVerilog is done during compile time which provides a better run time performance. SecVerilog's advantage over Caisson is that it supports fine-grained sharing of hardware resources across security levels.

Since SecVerilog also makes use of static IFT, the design flow is very similar to Caisson in Figure 5. (1) The designers first define a security-label policy. For example, suppose the security policy of a design has two security levels: *Low* and $High$. The adversary has access to all information at or below the *Low* security level, and can measure the clock cycles of hardware operations. $High$ labels the data that should be protected from such an adversary. (2) During the implementation of the design in Verilog, each variable has to be labeled with its corresponding security label, highlighted in Figure 8 . (4) Using these labels, the information flow is checked during compile time by SecVerilog, and any violations are highlighted.

```
1  reg[18:0] {L} tag0[256], tag1[256];
2  reg[18:0] {H} tag2[256], tag3[256];
3  wire[7:0] {L} index;
4  //Par(0)=Par(1)=L Par(2)=Par(3)=H
5  wire[1:0] {Par(way)} way;
6  wire[18:0] {Par(way)} tag_in;
7  wire {Par(way)} write_enable;
8
9  always @ (posedge clock) begin
10    if (write_enable) begin
11       case (way)
12       0: begin tag0[index] = tag_in; end
13       1: begin tag1[index] = tag_in; end
14       2: begin tag2[index] = tag_in; end
15       3: begin tag3[index] = tag_in; end
16       endcase
17    end
18 end
```

Fig. 8. Split cache in SecVerilog, modeled after [148]. Security labels are highlighted.

Labeling does not solve all the problems of information flow, especially if resources are shared. This is a limitation of static information flow tracking, e.g. in Caisson [89]. In the case of shared resources, the labels might change during runtime. SecVerilog use dependent types to handle runtime label changes. A design of split cache is shown in Figure 8 as an example. Type changes are detected and updated dynamically during the runtime, e.g. $Par(way)$ in Figure 8. The dependent types can be determined by type-valued functions: For a variable v, the type of the variable can be determined dynamically during runtime by a function, e.g. $Par(v)$, line 4 Figure 8.

A secure MIPS processor and caches were designed in SecVerilog. Dynamic labeling makes the shared ports of the cache possible. SecVerilog provides timing-sensitive non-interference, which is proved in the paper [148].

Table 1. Comparisons of Caisson, SecVerilog and Sapper

	Caisson	SecVerilog	Sapper
Type of Information Flow	Static (Compile time)	Static (Compile time)	Dynamic (Runtime)
Type of Label	Static Label	Dynamic Label	Run-time Label
Pipeline design	Time Mux	Time Mux	Time Mux
Cache design	Split Cache Split Ports	Split Cache Shared Ports	Split Cache Shared Ports

Comparison of Caisson, Sapper and SecVerilog A comparison of Caisson, Sapper, and SecVerilog is shown in the Table 1. Caisson and SecVerilog use static information flow control, while Sapper uses dynamic information flow control. Static information flow tracking is done at the design time, and does not introduce any runtime overhead for the system. Also, designers can fix information leakage at design time, and thus do not need to worry about the effects of security violations at runtime, since no violations will appear then. Compared to Caisson [89], dynamic labeling in SecVerilog and dynamic IFT in Sapper make resources sharing possible, thereby reducing area and timing overheads.

4.3 Security Verification using Model Checkers

Another approach to verify the security properties of hardware systems is using model checkers. First, the model of the system separated from system design should be built in the form of a finite state machine (FSM). Usually, the model is simplified to avoid the state explosion problem. The security properties are represented by a set of invariants. The model checker can automatically search for all possible states, and check if the invariants always hold. If so, the security properties are said to be proved.

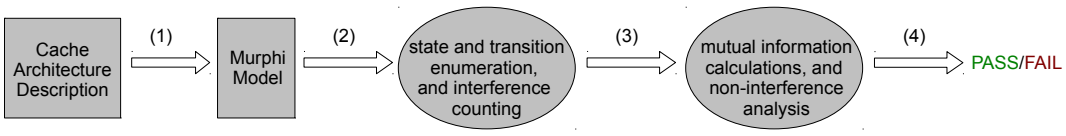

Fig. 9. Processor cache security verification Murφ model checker.

Cache Verification Processor caches are integral part of any modern processor. They are small, but fast memory components that are used to provide quick access to frequently accessed data. Through a fixed algorithm, the cache logic decides which data to keep in the cache and which data to send back to memory

if a new request comes and there is not sufficient space in the cache. Memory access timing changes depending on whether a request "hits" or "misses" in the cache. Based on this timing difference between hits and misses, researchers have presented numerous side-channel attacks, e.g., [94], that are able to compromise data confidentiality and potentially leak out cryptographic keys.

In [149] researchers create side-channel leakage models based on the non-interference property between an attacker and a victim process that are using same processor cache. Starting with an architecture description, (1) in Figure 9, they model the cache architecture as an FSM with states representing which process is currently using the cache line and transitions between the states based on cache operations (e.g. attacker cache hit, victim cache miss, etc.). By modeling the cache operation and transitions, the authors were able to obtain probabilities for how different operations of the victim (e.g. cache hit, cache miss, etc.) are observed by the attacker. For example, if cache line changes state (that attacker can observe) due to an action of the victim, then there is some interference, breaking the non-interference property. Zhang and Lee used Murφ to enumerate all possible states and transitions, and count the number of inferences between attacker and victim for the different state transitions, (2) and (3) in Figure 9. Based on this data, mutual information [32] is then used to quantitatively analyze the interference between the two processes, and reveal side-channel vulnerabilities, (4) in Figure 9.

The method is applied to six cache architectures and reveals that most cache architectures do not satisfy the non-interference property, thus fail the verification. To apply this method to other designs, designers need to manually create the Murφ system representation from the cache architecture description, as there is currently no automated way to extract these models from the system representation (e.g. from HDL code).

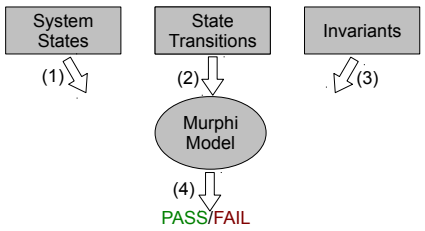

Fig. 10. Verifying XOM in Murφ Model Checker

XOM The eXecute Only Memory [91] is a hardware design with embedded cryptographic functionality and access control. By adding new hardware and new instructions, XOM is able to protect user data from a malicious operating system. On-chip data is isolated using hardware tags labeling the owner of data,

while off-chip data is protected by encryption and hashing. In [90], XOM was formally specified and then verified against an adversary in Murφ.

A model of XOM and its adversary is build in Murφ as in Figure 10. The model of the XOM hardware contains arrays representing the register, cache, and memory, including data and tags, (1) in Figure 10. The function of each instruction, which is the state transition function in Murφ, is defined according the design of XOM (2). To model the effect of the adversary, two identical sets of states are used, dubbed the "actual world" and the "ideal world". In the actual world, the adversary is modeled by a set of primitive actions she can perform as state transition functions, (2) in Figure 10. The ideal world does not include the effect of the adversary. The actual world states and ideal world states are concatenated, and thus updated together during model checking. With the model and state transition function, Murφ is able to exhaustively search for all possible combinations of these actions. Invariants are defined according to the security properties to be verified, (3). To prove the adversary is never able to read user data, the model checker verifies that the user data is either tagged with user XOM ID or encrypted and hashed with the user's key. To prove that the adversary cannot modify the user data without halting the system, the model checker compares the state of the ideal world against the state actual world, and thus, knows whether the adversary will succeed.

During verification, a replay attack was identified and fixed. Moreover, it was shown that if the operating system does not behave maliciously, and the liveness of the system is guaranteed.

Comparison of XOM and Cache Verification Cache Verification and XOM use a similar approach for verification using Murφ. The main difference is that Cache Verification concerned only about non-interference. XOM, on the other hand has a different model for detecting attacks by tagging each memory location with ID of the owner of the memory location. The security property being sought can be customized for XOM verification with the help of the tagging mechanism.

4.4 Commerical Tools

Even though they are relatively new, there is a number of commercial security verification tools for hardware design.

Mentor Graphics Questa Secure Check The application is part of Mentor Graphics Questa package. It receives RTL data and a spec for secure storage and paths. The spec is defined in TCL language. Secure Check then finds ports/black box inputs and generates properties for integrity and confidentiality. Black box inputs are generated in a way that it assures that no information flows outside of its designated path. The application then verifies these properties. The output of the application is an exhaustive proof of integrity and confidentiality of the design and/or counterexamples showing how your spec can be violated [51].

Cadence JasperGold Security Path Verification (SPV) App Similar to Questa Secure Check, SPV App takes RTL data and path specs. The user defines illegal sources and destinations of the data. SPV App proves that the defined secure data maintains confidentiality and integrity during operation and even after a hardware fault occurs. Verification is performed exhaustively using Jaspers path sensitization technology. Path sensitization technology utilizes the path cover property in which there is a source signal and a destination signal. By proving path cover property, the signal at the source of the path is tainted. The app formally verifies if it is possible to cover a tainted signal at the destination. When the property is covered, a waveform displays how data can propagate from source to destination. The property can also be determined to be unreachable, which means that it is not possible for data to propagate from source to destination. Verification can also be tuned by the user by creating black box modules where data can enter or not. This will simplify the process of verification to scale well [24].

5 Security Verification Focusing on Software Levels of a System

The second class of projects that we survey deals with works that verify security properties of software, while considering the ISA or a machine model of the hardware, typically at the microarchitecture level. Verification here considers some memory model, register files, and other components of the hardware that constitute the environment on which the code will run. Software security verification work that does not consider any hardware in the verification process is outside of the scope of this survey. For software-only security verification, we refer the reader to the following surveys [47, 12, 112, 130, 142].

5.1 Security Verification using Model Checkers and SMT Solvers

MOAT MOAT [131] proposed to find vulnerabilities in enclave user programs that run on Intel SGX architecture. To protect sensitive data and code from disclosure or modification by infrastructure attackers (e.g. malicious OS) or other malicious programs, Intel developed Software Guard Extensions(SGX) [5]. Intel SGX makes such protection possible by providing an isolated memory region called *enclave*. The hardware primitives provided by SGX enforce that only the code inside the enclave can access data within the enclave. However, it cannot protect an enclave user program from leaking sensitive information from within if the software running in the enclave is not programmed properly, thus the need for verification.

The input of MOAT is the x86+SGX assembly code, (1) in Figure 11, of an enclave user program, alongside with annotations that indicate: a) program points where secrets are generated (e.g. decryption) and b) memory locations where secrets stored. The usage of assembly code as input to the verification process eliminates the need for a trusted compiler. MOAT then translates the

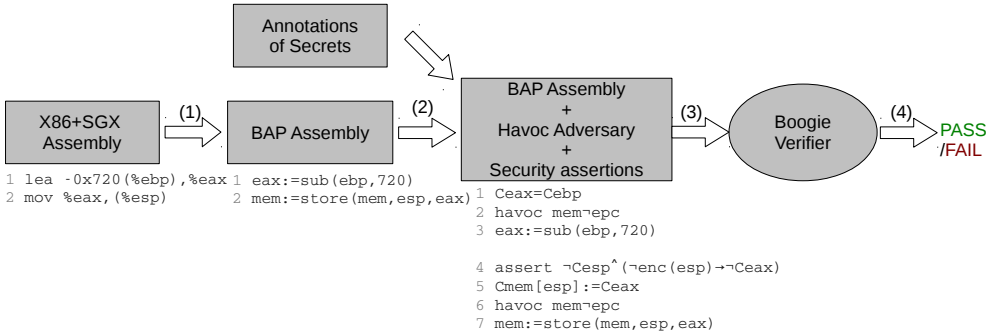

Fig. 11. MOAT verification.

assembly code to BAP (Binary Analysis Platform) assembly [21], as shown by (1) in Figure 11. MOAT uses BAP assembly for precise modeling of x86 and SGX instructions in Boogie verifier.

Inside MOAT, BAP assembly and the secret annotations are converted to code with assertions that Boogie can process. Two kinds of adversaries are considered: active adversaries who can write to any locations in non-enclave memory and passive adversaries who can read any location in non-enclave memory. To model the adversary, MOAT introduced a *havocing adversary*, who can modify all non-enclave memory locations between any consecutive statements of the enclave program and can observe all non-enclave memory. To show the effect of the adversary, a havoc instruction (havoc mem$\neg_{epc}$) is added before every BAP instruction, (2) in Figure 11. To reason about confidentiality, ghost variables (C_x) are also added. If C_x is true then the data x in registers or memory is dependent on a secret. Based on the value of the C_x, one can judge whether there is secret data leaking to non-enclave memory. E.g. line 4 of (2) in Figure 11, asserts data in $\%eax$ can be written to mem[$\%esp$] only if $\%esp$ does not depend on any secret (no control flow), and if $\%esp$ is in non-enclave memory ($\neg enc(esp)$) then $\%eax$ must not depend on a secret. This way, assumptions and assertions about the ghost variable are added, see (2) in Figure 11. The system's security assertions are verified by Boogie verifier (3). If the assertions do not always hold, then there is violation to confidentiality, and the verifier returns the violating piece of code, otherwise the design passes; see (4) in Figure 11.

MOAT provides a methodology to prove the security properties of software developed for the Intel SGX architecture. One-time password (OTP) service enclave program is verified as an example. It is also the first work to create formal a model of Intel's new SGX instructions.

Verve Verve [146] is an operating system that is verified to guarantee memory and type safety. Verve's architecture consists of two levels. The first level is called the "nucleus" that implements the core functionality needed to access memory and hardware. On top of nucleus, there is a kernel level which supports

functionality such as preemptive threads. The applications run on top of these two levels.

Verve uses two strategies to verify the nucleus and kernel. The nucleus is written in Boogie programming language and verified by Boogie. The code of nucleus is manually annotated with assertions that include preconditions, post-conditions and loop invariants. Some of the code is written in assembly and the assembly instructions from the nucleus code are also converted into Boogie so that they can have the annotations. The kernel ensures type safety using Typed Assembly Language (TAL) [108] and a TAL-checker [27]. The kernel is written in safe C# and the code is then compiled to TAL by a special compiler. TAL is used to verify that the assembly code does not violate the primitive abstractions of the language. TAL provides abstractions for basic types, labels, and pointers. Using these abstractions, invalid operations such as arithmetic operations on pointers are not allowed.

Verve is still an experimental OS which lacks some of the modern features such as exception handling and multiprocessor support. However, it supports type safety in the whole OS including the applications. It demonstrates that using automated techniques, high level code (such as safe C#) can be verified for type safety in assembly level using type-safe assembly languages (such as TAL).

Ironclad Apps Ironclad Apps is a project [60] focuses on the execution of remote applications in a secure and a functionally-verified manner. The verification process covers the code that is executed remotely, the remote OS, libraries, and drivers. Therefore, Ironclad Apps can be regarded as a multi-level verification system which assumes that the hardware is secure. Ironclad Apps eliminates data leaks and software based vulnerabilities. However, it is not designed for hardware-based attacks (side-channels, etc.) nor denial-of-service attacks.

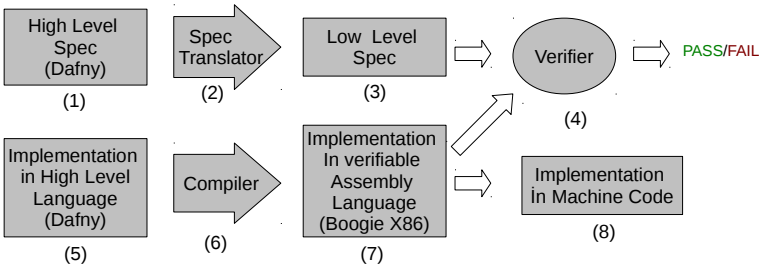

Fig. 12. Ironclad verification process.

Ironclad uses Verve as the OS. The verification process requires a high-level code and a high-level specification of the application code which are written in Dafny [85]. The spec and the code are handled in parallel. The code is compiled

to output assembly code in the BoogieX86 assembly language (note that the verifier Boogie and the assembly language BoogieX86 are different) [146], (5), (6), and (7) in Figure 12.

Meanwhile, the high-level spec is translated into a low-level spec by a spec-translator tool, (1), (2), and (3) in Figure 12. The low level spec and assembly code are then verified together to see if they are functionally equivalent and free of software vulnerabilities. If the verifier fails, the process has to be restarted with fixed code and spec. When verification passes successfully, the assembler and linker tools convert the assembly code into machine code, (8) in Figure 12.

Ironclad Apps is designed to protect against software-only attacks. The threat model assumes there might be some potentially malicious software running before Ironclad Apps starts and after it ends. The main hardware components are assumed to be trusted. However, the BIOS, and peripheral devices can be malicious. Even though the verification process is performed over low-level code, developers do not need to program in low-level code. As seen in the step labelled as (5) in Figure 12, the implementation language is Dafny [85]. Along with the actual implementation, the designers also generate a high-level specification, as seen in step (1). The verification is performed after Dafny code has been compiled into BoogieX86 assembly language as seen in steps (6), (7), and (4) in Figure 12.

SecVisor SecVisor [127, 46] is a hypervisor designed to provide execution and code integrity. It guarantees that only user-approved code can execute in kernel mode, and the approved code can only be modified by SecVisor. SecVisor leverages hardware memory protections and kernel privilege level to achieve execution and code integrity. The design assumes that the attacker has control of everything except the CPU, the memory controller, and the system memory. The small codebase makes the formal verification of SecVisor possible.

A model in Murφ is developed to verify the system. The model consists of three parts: a model of the hardware, a model of SecVisor, and a model of the attacker. Since the security of SecVisor is based on the hardware memory protections, it is crucial to specify the hardware model and the page table in the SecVisor model correctly. The hardware model includes physical memory, CPU mode bits, program counter, and a Device Exclusion Vector (DEV) that controls DMA permissions. The initialization, CPU mode transitions, and page-table synchronization in SecVisor are modeled in Murφ. To deal with the state space explosion problem, the authors model SecVisor conservatively. So when Murφ returns with success, the SecVisor is proved to satisfy all the security properties.

To model the attacker, an ideal model and an actual model are used, akin to the models used in the verification of XOM[90]. In the actual model, the attacker can write to any memory pages with the read/write bit set, or physical pages whose DEV bit is not set and also update kernel page tables. The execution integrity is the equality between the approved status of a page in the actual model and that in ideal model (without an attacker). The code integrity means

that the attacker cannot modify the approved code. The execution integrity and code integrity invariants are checked by the Murφ model checker.

XMHF XMHF (eXtensible and Modular Hypervisor Framework) [141] is designed to achieve a modular extensibility, automated verification, and high performance. The focus of verification is on memory integrity, which means memory regions can only be modified by the hypervisor. To verify the memory integrity, security invariants assertions are inserted into the C code. Most of the C code is verified automatically by CBMC model checker, while the remaining is manually audited.

5.2 Security Verification using Formal Proofs

seL4 [75] was the first OS that was formally verified for functional correctness. Even though this survey focuses on security verification, we want to mention the effort that has been done for seL4, since it sheds some light on how verification can be handled at the system level. The aim of the seL4 verification effort is to provide a system free of programming errors that introduce vulnerabilities that may cause failures or facilitate attacks. seL4 is a software-only work and assumes that the underlying hardware, the compiler, and the low-level device driver code are provided free of errors. Under these circumstances, seL4 guarantees that it will never crash and never lead to an unsafe operation that can be exploited.

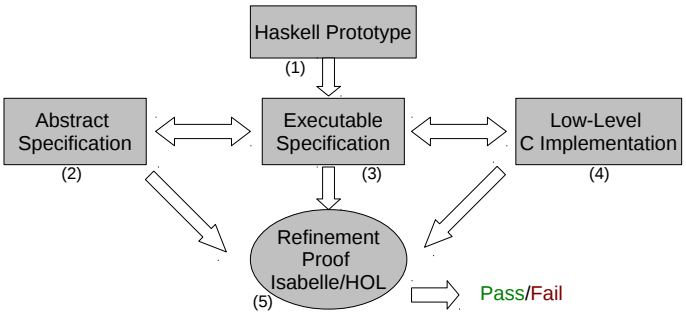

Fig. 13. The seL4 verification process.

seL4 operating system is based on L4 [92] and features a virtual-address space, threading, inter-process communication, and capabilities for authorization. The designers of seL4 follow an approach well-suited for cooperation between system designers and system verifiers. They use Haskell as OS-design prototype language. However, while the Haskell prototype is used for higher-level verification, the actual low-level implementation does not use Haskell. The programmers manually reproduce the same model in C. They follow this approach

because Haskell code is rather unsuitable for low-level functions performance-wise, and because C is easier to verify. For formal verification, the Isabelle/HOL theorem prover is used interactively. The property of the proof they used is a refinement which establishes a one-to-one correspondence between the higher-level and lower-level representations of the system. If the system has a security property in the higher level, refinement guarantees that it would be preserved in the kernel code. This approach makes both specifying the proofs and proving them more feasible in the higher levels. The same work at the kernel level would require much more time and effort.

The verification process is depicted in Figure 13. The system prototype is coded in Haskell according to a high-level specification, (1) in Figure 13. The specification includes a detailed functional and behavioral description of the system, (2) in the figure. Isabelle/HOL theorem prover generates an Executable Specification out of the Haskell code, (3) in the figure. This process is critical since it will directly impact the correctness of the system, any misrepresentations can render the verification ineffective. This specification contains all data structures and implementation details that the low-level implementation must have. The last layer is the actual C implementation of seL4, (4) in Figure. These three layers – abstract specification, executable specification, and C implementation – are the refinement layers used in the formal verification, (5) in Figure 13.

CertiKOS and Deep Specifications [53] presents a design technique based on modern computer system architectures (such as OSes) where each system consists of abstraction levels such as kernels, hypervisors, device drivers, network protocols. Each layer defines an interface and hides the implementation. Deep Specifications is based on the verification of abstraction layers that define interfaces to other layers hiding the implementation details.

Deep specification has been used to implement four variants of mCertiKOS kernel. In mCertiKOS, each layer represents an abstraction, and its behavior is defined in a specification as shown in Figure 14. These specifications are called deep specifications and any two implementations that have the same deep specification must have contextually equal behavior regardless of the implementation method. Hence, mCertiKOS relies on deep specification of layers rather than their specific implementations and as long as an implementation of an abstraction layer can be proven to be equivalent to its corresponding deep specification, it can be used without violating the general correctness of the system. An error-free and functionally-correct implementation of the whole system relies on implementing the abstraction layers correctly. Unlike seL4 [75], where the whole system is verified at once, mCertiKOS can be verified layer by layer or as a whole.

mCertiKOS uses two core languages for high-level and assembly-level code: ClightX and LAsm. ClightX is based on CompCert Clight language [19], and LAsm is an assembly language customized for CertiKOS development. These languages can be used to implement abstraction layers. The layer interfaces and Deep Specifications are described using Coq. mCertiKOS uses the CompCertX

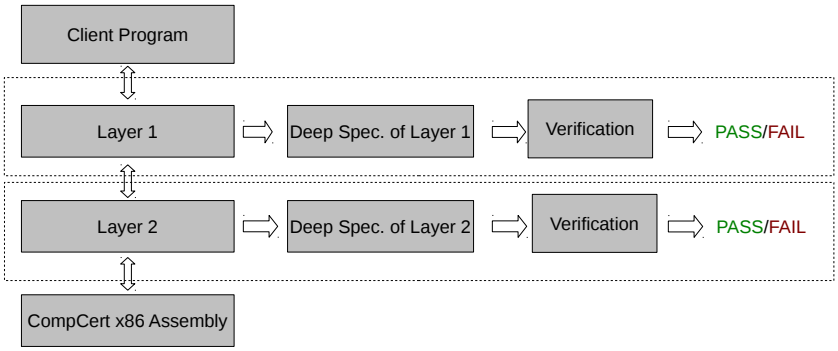

Fig. 14. CertiKOS verification process.

compiler for both languages. CompCertX is a specialized version of CompCert compiler that works with the mCertiKOS memory and machine model. If implementations M1 and M2 implement the same DeepSpec, they should have contextually equivalent behavior. Three other versions of mCertiKOS (mCertiKOS-hyp, mCertiKOS-rz and mCertiKOS-emb) [53] are implemented to demonstrate how additional layers or different implementations can be added to the kernel. mCertiKOS-hyp kernel provides support for user-level hypervisors based on AMD SVM [3] technology. mCertiKOS-rz is implemented to demonstrate how existing layers can be augmented with the support of ring 0 processes. mCertiKOS-emb is a minimalist kernel where memory protection and system call interfaces are eliminated. It shows how layers can be eliminated and the whole system still be verified using DeepSpec.

MinVisor MinVisor [98] is a simple hypervisor, which protects its own memory from malicious guests. This work was presented as a follow-up work on SecVisor, but using theorem proving approach. The goal of the project is to fully verify the MinVisor at the assembly level using ACL2. A series of detailed and accurate models of the AMD64 instruction set architecture (ISA), including the memory model, registers, and state transitions, were developed. Several theorems, such as the one where isolation of model specific registers and MinVisor memory are guaranteed against guest modifications, are proved to show the security properties of MinVisor.

6 Analysis of Existing Work

Tables 2 and 3 present a summary of the main projects reviewed in Sections 4 and 5. In these tables, we compare the existing works in terms of their verification methods, the levels they consider in a system, and the security aspects being verified.

6.1 Verification Methods

Most of the projects use general purpose verification tools, as shown in the *Tool* column of Tables 2 and 3. The current general-purpose tools used in security verification are not compatible with conventional hardware or software languages, such as C or Verilog, and verification is performed as an additional step after design and implementation. Security specifications are described as formulas in theorem provers like Coq, or as invariants in model checkers like Murφ, as shown in the *Specification* column of Tables 2 and 3. A model (system representation) separate from the system implementation (actual system) is built, e.g. Micro-Policies [39], Cache verification [149], XOM [91], SecVisor [46]. Designers have to make sure their model accurately mirrors the system implementation, otherwise the result of verification might be not correct.

Some projects take approach of designing new domain-specific languages that allow making verification an integral part of the design and implementation process. In these projects, tools are developed to transform the system description in the new domain-specific language into another form that is amenable to use with verification tools, e.g. VeriCoq, Dafny, or TAL-compiler. For example, in Dafny, the code has annotations for pre- and post-conditions, invariants, and ghost variables. With use of annotations and through automatic transformation SMT solvers can check if the invariants always hold. Meanwhile, other projects embed security-related tags into a conventional language, and facilitate describing the security-properties to be verified. These projects tend to develop custom tools, as shown in Tables 2 and 3 to make sure the generated design has the desired security properties, such as Sapper [88], Caisson [89], and SecVerilog [148].

Designers should decide which approach to take for their development cycle. On the one hand, they can develop their system in a "traditional" language. This allows for quick development of the functional design with tools familiar to engineers, but does incur the effort of having to also separately write their design in a representation that their preferred security verifier understands. On the other hand, they can implement their system in a verification-friendly language. This has higher initial effort, but may pay off in long term with less effort due to not having to write the representation second time for verification. The drawback is that the verification-friendly language may not support all the aspects the designer desires to verify.

6.2 Verification Aspects

Confidentiality and integrity are the two main security properties often sought in a system. The verification aspect of a system often covers these properties, but can be formulated in a more generic form (e.g. non-interference) or a more specific form (e.g. memory integrity). The formulation of these properties depends on the levels that the system spans, and on the tools used. The analysis of information flow provides a useful basis for proving these security properties of a system. Monitoring information flow requires data labeling, declassification, and

information flow rules specific to the system. We observe that many hardware projects use the analysis of information flow for proving information flow policies, non-interference, and confidentiality and integrity, as seen in Table 2. Software projects, as illustrated in Table 3, have a wider variety of verification aspects, which try to verify confidentiality and/or integrity, but only within the selected levels. Designers generally try to provide partial integrity or confidentiality for a system. For example, SecVisor [46] verifies execution and code integrity which is a subsection of the whole memory.

Designers should decide which security aspects to prove in their design, that suit well with both implementation and verification of the system at the same time.

Table 2. Summary of projects that focus on hardware verification. These projects were detailed in Section 4.

Name	Specification	Tool (Sec. 3)	Custom Tool	Levels								Ver. Method (Sec. 3)	Ver. Aspect (Sec. 2)
				App	OS	Hypervisor	ISA	uArch	RTL	Gate	Physical		
VeriCoq [17]	Spec. in Coq Lang. + Verilog	Coq IDE	VeriCoq					✓	✓			Theorem Prover	Information Flow
Micro-Policies [39]	Spec. in Coq Lang.	Coq IDE	none				✓					Theorem Prover	Information Flow
Caisson [89]	Cassion Language	none	Cassion Tool					✓	✓			Pen-and-paper Proof	Non-interference
Sapper [88]	Sapper Language	none	Sapper Tool					✓	✓			Pen-and-paper Proof	Non-interference
SecVerilog [148]	SecVerilog Language	none	SecVerilog Tool					✓	✓			Pen-and-paper Proof	Non-interference
Cache Verification [150, 149]	Murφ Language	Murφ	none					✓				Model Checking	Confidentiality + Integrity
XOM [91]	Murφ Language	Murφ	none					✓	✓			Model Checking	Confidentiality + Integrity

Table 3. Summary of projects that focus on software verification with respect to the ISA or machine model. These projects were detailed in Section 5.

Name	Spec	Tool (Sec. 3)	Custom Tool	Levels								Ver. Method (Sec. 3)	Ver. Aspect (Sec. 2)
				App	OS	Hypervisor	ISA	uArch	RTL	Gate	Physical		
MOAT [131]	Assembly + Annotations	Boogie, BAP, Z3	none	✓			✓					SMT solvers	Confidentiality
Verve [146]	Typed Assembly + Annotations + C#	Boogie, TAL Checker	none		✓		✓					SMT solvers	Type and Memory Safety
Ironclad Apps [60]	High-level Spec.	Dafny, Boogie, Z3	Custom Compiler + Spec. Translator	✓			✓					SMT solvers	Memory Safety + Functional correctness
SecVisor [46]	High Level Spec	Murϕ	none			✓		✓				Model Checking	Execution and code Integrity
XMHF [141]	C	CBMC	none			✓	✓	✓				Model Checking	Memory Integrity
CetriKOS [53]	Clight + LAsm	Coq IDE	none		✓	✓	✓					Theorem Prover	Non-interference + Functional Correctness
SeL4 [75]	C + Haskell	Isabelle/HOL	none		✓							Theorem Prover	Functional Correctness
MinVisor [98]	Assembly	ACL2	none			✓	✓	✓				Theorem Prover	Code Integrity

6.3 Verification Levels

The Trusted Computing Base (TCB) often encompasses multiple levels of the system from hardware to software. However, as can be seen in Tables 2 and 3, verification projects are typically focused on the hardware levels, or focused on the software levels. Bringing the hardware and software levels together is difficult, however, needed. For example, enhancing the security of software levels by using support in hardware levels is becoming a more viable approach, especially for remote computing. Hardware based TCBs are emerging quite rapidly, such as ARM TrustZone [140], Intel SGX [100], and AMD memory encryption[4]. The working of this hardware with software needs to be verified for security, and requires spanning many system levels.

Designers should consider expanding their approaches to include more levels into security verification, to allow truly full-system verification.

7 Conclusion

Formal verification research has been mostly focused on the functional correctness of the hardware or software systems. Security verification of software-only is also well studied. Hardware security verification, however, is an emerging research area which is necessitated by the fact that modern systems require both software and hardware for their correct and secure operation. Especially with introduction of security-focused hardware, such as Intel SGX. Trusting remote software and hardware is more critical now than before, as it handles users' even-increasing sensitive information. Any vulnerabilities in these computing systems can be exploited by attackers. Thus, the whole system, including both the hardware and software parts, should be considered in the security verification.

Security verification is a branch of formal verification where the correctness properties are extended to include security properties, e.g. confidentiality and integrity. The process requires a formal, mathematical specification of the security properties, an accurate representation of the implementation, and some verification mechanisms, e.g. theorem proving and model checking, to prove that the implementation complies with the needed security properties.

In this survey, we focused on the security verification projects that involve at least some hardware and software levels. Since security properties are provided by multiple levels in the system, only verifying some particular level or levels cannot guarantee whole system's security. With the improvement of verification tools and methods, as presented in this survey, there is a trend to include more and more system levels in verification, but not yet all levels. We provide an insight into the tools and mechanisms used for security verification, and compare projects based on security verification of hardware and software levels they consider.

There are many open research topics in the security verification of hardware and software systems. The most critical, however, is the need for full-system security verification, which spans more levels than can be done through today's existing projects.

Acknowledgement

This work is supported in part by the National Science Foundation (NSF) grants 1419869 and 1524680; and Semiconductor Research Corporation (SRC) contract 2015-TS-2633. Dr. Demir's work is supported by TUBITAK grant 2219.

References

1. Abrial, J.R., Lee, M.K., Neilson, D., Scharbach, P., Sørensen, I.H.: The b-method. In: VDM'91 Formal Software Development Methods. pp. 398–405. Springer (1991)
2. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from c protocol code by symbolic execution. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 331–340. CCS, ACM (2011)
3. AMD: Amd64 architecture programmers manual volume 3: General-purpose and system instructions. Tech. rep. (2013)
4. AMD: AMD Memory Encryption (2016), http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed May 2016
5. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for cpu based attestation and sealing. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (2013)
6. Armstrong, A., Struth, G., Weber, T.: Program analysis and verification based on kleene algebra in isabelle/hol. In: Interactive Theorem Proving, pp. 197–212. Springer (2013)
7. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K.: Chisel: constructing hardware in a scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference. pp. 1216–1225. ACM (2012)
8. Baier, C., Katoen, J.P., et al.: Principles of model checking. MIT press Cambridge (2008)
9. Barrett, C., Fontaine, P., Tinelli, C.: The smt-lib standard version 2.6 (2010)
10. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Proceedings of CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer (2009)
11. Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Meier, A., Melis, E., et al.: mega: Towards a mathematical assistant. In: Automated DeductionCADE-14, pp. 252–255. Springer (1997)
12. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and software verification: model-checking techniques and tools. Springer Science & Business Media (2013)
13. Berdine, J., Bjørner, N.: Computing all implied equalities via smt-based partition refinement. In: Automated Reasoning, pp. 168–183. Springer (2014)
14. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions. Springer Science & Business Media (2013)
15. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: A verifier for gpu kernels. SIGPLAN Not. 47(10), 113–132 (Oct 2012)
16. Bidmeshki, M.M., Makris, Y.: Toward automatic proof generation for information flow policies in third-party hardware ip. In: International Symposium on Hardware Oriented Security and Trust (HOST). pp. 163–168 (May 2015)
17. Bidmeshki, M.M., Makris, Y.: Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In: International Symposium on Circuits and Systems (ISCAS). pp. 29–32. IEEE (2015)
18. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: Proceedings of ICCAD 2008. pp. 356–363. IEEE (2008)

19. Blazy, S., Leroy, X.: Mechanized semantics for the clight subset of the c language. Journal of Automated Reasoning 43(3), 263–288 (2009)
20. Blech, J.O., Gesellensetter, L., Glesner, S.: Formal verification of dead code elimination in isabelle/hol. In: International Conference on Software Engineering and Formal Methods. pp. 200–209. IEEE (2005)
21. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A Binary Analysis Platform, pp. 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
22. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Proceedings of TACAS 2007. Lecture Notes in Computer Science, vol. 4424, pp. 358–372. Springer (2007)
23. Buchberger, B., Crciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., et al.: Theorema: Towards computer-aided mathematical theory exploration. Journal of Applied Logic 4(4), 470–504 (2006)
24. CADENCE: Jaspergold security path verification app (2016), url-http://www.cadence.com/products/fv/jaspergold_security/pages/default.aspx
25. Carter, W.C., Joyner Jr, W.H., Brand, D.: Symbolic simulation for correct machine design. In: Conference on Design Automation. pp. 280–286. IEEE (1979)
26. Champagne, D., Lee, R.B.: Scalable architectural support for trusted software. In: International Symposium on High Performance Computer Architecture (HPCA). pp. 1–12. IEEE (2010)
27. Chen, J., Hawblitzel, C., Perry, F., Emmi, M., Condit, J., Coetzee, D., Pratikaki, P.: Type-preserving compilation for large-scale optimizing object-oriented compilers. In: ACM SIGPLAN Notices. vol. 43, pp. 183–192. ACM (2008)
28. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: A platform for in-vivo multi-path analysis of software systems, vol. 47. ACM (2012)
29. Chou, C.T., Mannava, P.K., Park, S.: Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings, chap. A Simple Method for Parameterized Verification of Cache Coherence Protocols, pp. 382–398. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
30. Clarke, E., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: A decade of concurrency reflections and perspectives, pp. 124–175. Springer (1993)
31. Clarke, E., Jha, S., Marrero, W.: A machine checkable logic of knowledge for specifying security properties of electronic commerce protocols. In: Workshop on Formal Methods and Security Protocols (1998)
32. Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2012)
33. Crary, K.: Toward a foundational typed assembly language. ACM (2003)
34. Cresswell, M.J., Hughes, G.E.: A new introduction to modal logic. Routledge (2012)
35. Crow, J., Di Vito, B.: Formalizing space shuttle software requirements: Four case studies. ACM Trans. Softw. Eng. Methodol. 7(3), 296–332 (Jul 1998)
36. Cui, H., Hu, G., Wu, J., Yang, J.: Verifying systems rules using rule-directed symbolic execution. SIGPLAN Not. 48(4), 329–342 (Mar 2013)
37. Daum, M., Dörrenbächer, J., Schmidt, M., Wolff, B.: A verification approach for system-level concurrent programs. In: Verified Software: Theories, Tools, Experiments, pp. 161–176. Springer (2008)

38. Daum, M., Schirmer, N.W., Schmidt, M.: Implementation correctness of a real-time operating system. Third IEEE International Conference on Software Engineering and Formal Methods (SEFM) pp. 23–32 (2009)
39. De Amorim, A.A., Dénes, M., Giannarakis, N., Hritcu, C., Pierce, B.C., Spector-Zabusky, A., Tolmach, A.: Micro-policies: Formally verified, tag-based security monitors. In: IEEE Symposium on Security and Privacy (SP). pp. 813–830. IEEE (2015)
40. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
41. Dhawan, U., Vasilakis, N., Rubin, R., Chiricescu, S., Smith, J.M., Knight Jr, T.F., Pierce, B.C., DeHon, A.: Pump: a programmable unit for metadata processing. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy (2014)
42. Dill, D.L.: The mur ϕ verification system. In: Computer Aided Verification. pp. 390–393. Springer (1996)
43. Dwoskin, J.S., Lee, R.B.: Hardware-rooted trust for secure key management and transient trust. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 389–400. ACM (2007)
44. Engineering, N., Center, S.: Technical support to the national highway traffic safety administration (nhtsa) on the reported toyota motor corporation (tmc) unintended acceleration (ua) investigation. Tech. rep., NASA Engineering and Safety Center Technical Assessment Report (2011)
45. Farmer, W.M., Guttman, J.D., Thayer, F.J.: Imps: An interactive mathematical proof system. Journal of Automated Reasoning 11(2), 213–248 (1993)
46. Franklin, J., Seshadri, A., Qu, N., Chaki, S., Datta, A.: Attacking, repairing, and verifying secvisor: A retrospective on the security of a hypervisor. Tech. rep., Technical Report CMU-CyLab-08-008, Carnegie Mellon University (2008)
47. Garcıa-Ferreira, I., Laorden, C., Santos, I., Bringas, P.G.: A survey on static analysis and model checking. In: International Joint Conference SOCO. p. 443 (2014)
48. Gluck, P.R., Holzmann, G.J.: Using spin model checking for flight software verification. In: Aerospace Conference Proceedings, 2002. IEEE. vol. 1, pp. 1–105. IEEE (2002)
49. Gordon, M.: From lcf to hol: a short history. In: Proof, language, and interaction. pp. 169–185. MIT Press (2000)
50. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning 3(2), 153–245 (2010)
51. Graphics, M.: Mentor graphics questa secure check (2016), https://www.mentor.com/products/fv/questa-secure-check
52. Greve, D., Wilding, M., Vanfleet, W.M.: A separation kernel formal security policy. In: Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (2003)
53. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: ACM SIGPLAN Notices. vol. 50, pp. 595–608. ACM (2015)
54. Hallgren, T.: Alfa web site (2012), http://www.cs.chalmers.se/ hallgren/Alfa/
55. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM (JACM) 40(1), 143–184 (1993)
56. Hatnik, U., Altmann, S.: Using modelsim, matlab/simulink and ns for simulation of distributed systems. In: International Conference on Parallel Computing in Electrical Engineering. pp. 114–119. IEEE (2004)

57. Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W., White, J., et al.: Formal analysis of the remote agent before and after flight. In: Proceedings of the 5th NASA Langley Formal Methods Workshop. vol. 134 (2000)
58. Havelund, K., Lowry, M., Penix, J.: Formal analysis of a space-craft controller using spin. Software Engineering, IEEE Transactions on 27(8), 749–765 (2001)
59. Havelund, K., Pressburger, T.: Model checking java programs using java pathfinder. International Journal on Software Tools for Technology Transfer 2(4), 366–381 (2000)
60. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI). pp. 165–181 (2014)
61. Hazelhurst, S., Seger, C.J.H.: Symbolic trajectory evaluation. In: Formal Hardware Verification, pp. 3–78. Springer (1997)
62. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall, Englewood Cliffs, N.J. (1991)
63. Holzmann, G.J.: The model checker spin. IEEE Transactions on software engineering 23(5), 279 (1997)
64. Holzmann, G.J.: Mars code. Communications of the ACM 57(2), 64–73 (2014)
65. Holzmann, G.J., Smith, M.H.: Automating software feature verification. Bell Labs Technical Journal 5(2), 72–87 (2000)
66. Hua, W., Li, X., Guan, Y., Shi, Z., Zhang, J., Dong, L.: Formal verification for spacewire communication protocol based on environment state machine. In: International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM). pp. 1–4 (2012)
67. Huet, G., Kahn, G., Paulin-Mohring, C.: The coq proof assistant a tutorial. Rapport Technique 178 (1997)
68. 4th gen core family desktop specification update (2016), http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf
69. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: Tracer: A symbolic execution tool for verification. In: Computer Aided Verification. pp. 758–766. Springer (2012)
70. Jin, S., Ahn, J., Cha, S., Huh, J.: Architectural support for secure virtualization under a vulnerable hypervisor. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 272–283. ACM (2011)
71. Kaufmann, M., Moore, J.S.: An acl2 tutorial. In: Theorem Proving in Higher Order Logics, pp. 17–21. Springer (2008)
72. Keene, S.E.: A programmer's guide to object-oriented programming in Common LISP. Addison-Wesley Longman Publishing Co., Inc. (1988)
73. Kesten, Y., Pnueli, A., Raviv, L.o.: Algorithmic verification of linear temporal logic specifications. In: Automata, Languages and Programming, pp. 1–16. Springer (1998)
74. Kgil, T., Falk, L., Mudge, T.: Chiplock: support for secure microarchitectures. ACM SIGARCH Computer Architecture News 33(1), 134–143 (2005)
75. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al.: sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 207–220. ACM (2009)
76. Kreitz, C.: Building reliable, high-performance networks with the nuprl proof development system. Journal of Functional Programming 14(01), 21–68 (2004)

77. Kroening, D., Sharygina, N.: Approximating predicate images for bit-vector logic. In: Proceedings of TACAS 2006. Lecture Notes in Computer Science, vol. 3920, pp. 242–256. Springer Verlag (2006)
78. Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014)
79. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Proceedings of CAV 2006. Lecture Notes in Computer Science, vol. 4144, pp. 152–165. Springer Verlag (2006)
80. Langevelde, I.v., Romijn, J., Goga, N.: Founding firewire bridges through promela prototyping. In: Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS), 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA). IEEE Computer Society Press (2003)
81. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of standard ml. In: ACM SIGPLAN Notices. vol. 42, pp. 173–184. ACM (2007)
82. Lee, R.B.: Security basics for computer architects. Synthesis Lectures on Computer Architecture 8(4), 1–111 (2013)
83. Lee, R.B., Kwan, P., McGregor, J.P., Dwoskin, J., Wang, Z.: Architecture for protecting critical secrets in microprocessors. In: Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on. pp. 2–13. IEEE (2005)
84. Leinenbach, D., Petrova, E.: Pervasive compiler verification–from verified programs to verified systems. Electronic Notes in Theoretical Computer Science 217, 23–40 (2008)
85. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 348–370. Springer (2010)
86. Leino, K.R.M.: Developing verified programs with dafny. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 1488–1490. ICSE '13, IEEE Press (2013)
87. Leroy, X.: The compcert c verified compiler. Documentation and users manual. INRIA Paris-Rocquencourt (2012)
88. Li, X., Kashyap, V., Oberg, J.K., Tiwari, M., Rajarathinam, V.R., Kastner, R., Sherwood, T., Hardekopf, B., Chong, F.T.: Sapper: A language for hardware-level security policy enforcement. In: ACM SIGARCH Computer Architecture News. vol. 42, pp. 97–112. ACM (2014)
89. Li, X., Tiwari, M., Oberg, J.K., Kashyap, V., Chong, F.T., Sherwood, T., Hardekopf, B.: Caisson: a hardware description language for secure information flow. In: ACM SIGPLAN Notices. vol. 46, pp. 109–120. ACM (2011)
90. Lie, D., Mitchell, J., Thekkath, C.A., Horowitz, M.: Specifying and verifying hardware for tamper-resistant software. In: Symposium on Security and Privacy. pp. 166–177. IEEE (2003)
91. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. ACM SIGPLAN Notices 35(11), 168–177 (2000)
92. Liedtke, J.: Toward real microkernels. Communications of the ACM 39(9), 70–77 (1996)
93. Lipsett, R., Schaefer, C.F., Ussery, C.: VHDL: Hardware description and design. Springer Science & Business Media (2012)

94. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: IEEE Symposium on Security and Privacy. pp. 605–622 (2015)
95. Lockhart, D., Batten, C.: Hardware generation languages as a foundation for credible, reproducible, and productive research methodologies. In: Workshop on Reproducible Research Methodologies (REPRODUCE) (2014)
96. Luo, Z., Pollack, R.: LEGO proof development system: User's manual. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science (1992)
97. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in expressos. In: ACM SIGPLAN Notices. vol. 48, pp. 293–304. ACM (2013)
98. McCoyd, M., Krug, R.B., Goel, D., Dahlin, M., Young, W.: Building a hypervisor on a formally verifiable protection layer. In: System Sciences (HICSS), 2013 46th Hawaii International Conference on. pp. 5069–5078. IEEE (2013)
99. McCune, W., Shumsky, O.: Ivy: A preprocessor and proof checker for first-order logic. In: Computer-Aided reasoning, pp. 265–281. Springer (2000)
100. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. ACM (2013)
101. McMillan, K.L.: The smv system. In: Symbolic Model Checking, pp. 61–85. Springer (1993)
102. Megill, N.: Metamath. In: The Seventeen Provers of the World, pp. 88–95. Springer (2006)
103. Milner, R.: The definition of standard ML: revised. MIT press (1997)
104. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using murφ. In: Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on. pp. 141–151. IEEE (1997)
105. Mitchell, J.C., Shmatikov, V., Stern, U.: Finite-state analysis of ssl 3.0. In: USENIX Security (1998)
106. Mooij, A.J., Goga, N., Wesselink, W., Bosnacki, D.: An analysis of medical device communication standard ieee 1073.2. Communication Systems and Networks pp. 74–79 (2003)
107. Moore, J.S., Lynch, T.W., Kaufmann, M.: A mechanically checked proof of the amd5 k 86 tm floating-point division program. Computers, IEEE Transactions on 47(9), 913–926 (1998)
108. Morrisett, G., Walker, D., Crary, K., Glew, N.: From system f to typed assembly language. ACM Transactions on Programming Languages and Systems (TOPLAS) 21(3), 527–568 (1999)
109. Nikhil, R.: Bluespec system verilog: efficient, correct rtl from high level specifications. In: Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on. pp. 69–70. IEEE (2004)
110. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
111. Owre, S., Rushby, J.M., Shankar, N.: Pvs: A prototype verification system. In: Automated DeductionCADE-11, pp. 748–752. Springer (1992)
112. Paulin-Mohring, C.: Introduction to the coq proof-assistant for practical software verification. In: Tools for Practical Software Verification, pp. 45–95. Springer (2011)

113. Paulson, L.C.: Isabelle: A generic theorem prover, vol. 828. Springer Science & Business Media (1994)
114. Pfenning, F., Schürmann, C.: System description: Twelfa meta-logical framework for deductive systems. In: Automated DeductionCADE-16, pp. 202–206. Springer (1999)
115. van de Pol, J.C.: Automated verification of nested dfs. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 181–197. Springer (2015)
116. Rausch, M., Krogh, B.H.: Formal verification of plc programs. In: American Control Conference. vol. 1, pp. 234–238 (1998)
117. Romijn, J.: Improving the quality of protocol standards: Correcting IEEE 1394.1 firewire net update. Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatici 8, 23–30 (2004)
118. Russinoff, D., Kaufmann, M., Smith, E., Sumners, R.: Formal verification of floating-point rtl at amd using the acl2 theorem prover. Proceedings of the 17th IMACS World Congrress on Scientific Computation, Applied Mathematics and Simulation, Paris, France (2005)
119. Russinoff, D.M.: A case study in formal verification of register-transfer logic with acl2: The floating point adder of the amd athlon tm processor. In: Formal Methods in Computer-Aided Design. pp. 22–55. Springer (2000)
120. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on 21(1), 5–19 (2003)
121. Saudel, F., Salwan, J.: Triton: A dynamic symbolic execution framework. In: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015. pp. 31–54. SSTIC (2015)
122. Schirmer, N., et al.: Verification of sequential imperative programs in Isabelle-HOL. Ph.D. thesis, Technical University Munich (2006)
123. Schneider, F., Easterbrook, S.M., Callahan, J.R., Holzmann, G.J.: Validating requirements for fault tolerant systems using model checking. In: Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on. pp. 4–13. IEEE (1998)
124. Schneier, B.: The internet of things will turn large-scale hacks into real world disaster (2016), https://motherboard.vice.com/read/the-internet-of-things-will-cause-the-first-ever-large-scale-internet-disaster, accessed July 25, 2016.
125. Schürmann, C.: The twelf proof assistant. In: Theorem Proving in Higher Order Logics, pp. 79–83. Springer (2009)
126. Schwichtenberg, H., et al.: Proof-and program-extraction system minlog. Free code and documentation at http://www. minlog-system. de (2006)
127. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. ACM SIGOPS Operating Systems Review 41(6), 335–350 (2007)
128. Shacham, O., Wachs, M., Danowitz, A., Galal, S., Brunhaver, J., Qadeer, W., Sankaranarayanan, S., Vassiliev, A., Richardson, S., Horowitz, M.: Avoiding game over: Bringing design to the next level. In: Proceedings of the 49th Annual Design Automation Conference. pp. 623–629. ACM (2012)
129. Shmatikov, V., Mitchell, J.C.: Analysis of a fair exchange protocol. In: Network & Distributed System Security Symposium (2000)
130. Silva, V.D., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 27(7), 1165–1178 (2008)

131. Sinha, R., Rajamani, S., Seshia, S., Vaswani, K.: Moat: Verifying confidentiality of enclave programs. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1169–1184. CCS '15, ACM (2015)
132. SPIN: (2016), http://spinroot.com/spin/whatispin.html
133. Strecker, M.: Formal verification of a java compiler in isabelle. In: International Conference on Automated Deduction. pp. 63–77. Springer (2002)
134. Subramanyan, P., Malik, S., Khattri, H., Maiti, A., Fung, J.: Verifying information flow properties of firmware using symbolic execution. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 337–342. IEEE (2016)
135. Suh, G.E., Clarke, D., Gassend, B., Van Dijk, M., Devadas, S.: Aegis: architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th annual international conference on Supercomputing. pp. 160–171. ACM (2003)
136. Szefer, J., Lee, R.B.: Architectural support for hypervisor-secure virtualization. In: ACM SIGPLAN Notices. vol. 47, pp. 437–450. ACM (2012)
137. Thévenon, P.: Validation of proofs using phox. Electronic Notes in Theoretical Computer Science 140, 55–66 (2005)
138. Thomas, D., Moorby, P.: The Verilog® Hardware Description Language. Springer Science & Business Media (2008)
139. Thompson, K.: Reflections on trusting trust. Communications of the ACM 27(8), 761–763 (1984)
140. Trustzone, A.: Trustzone information page. Tech. rep. (2016), http://www.arm.com/products/processors/technologies/trustzone/
141. Vasudevan, A., Mccune, J.M., Newsome, J., Mellon, C.C.: Design and implementation of an extensible and modular hypervisor framework (2012)
142. Wallace, D.R., Fujii, R.U.: Software verification and validation: an overview. Ieee Software 6(3), 10 (1989)
143. Wenzel, M., et al.: Isabelle/Isara versatile environment for human-readable formal proof documents. Ph.D. thesis, Institut für Informatik, Technische Universität München. http://tumb1. biblio. tu-muenchen. de/publ/diss/in/2002/wenzel. html (2002)
144. Wiedijk, F.: The seventeen provers of the world: Foreword by Dana S. Scott, vol. 3600. Springer (2006)
145. Woodruff, J., Watson, R.N., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R., Roe, M.: The cheri capability model: Revisiting risc in an age of risk. In: Proceeding of the 41st annual international symposium on Computer architecuture. pp. 457–468. IEEE Press (2014)
146. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: ACM Sigplan Notices. vol. 45, pp. 99–110. ACM (2010)
147. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. Commun. ACM 54(12), 123–131 (2011)
148. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: ACM SIGARCH Computer Architecture News. vol. 43, pp. 503–516. ACM (2015)
149. Zhang, T., Lee, R.B.: New models of cache architectures characterizing information leakage from cache side channels. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 96–105. ACM (2014)
150. Zhang, T., Lee, R.B.: Secure cache modeling for measuring side-channel leakage. Tech. rep., Tech. Report, http://palms. ee. princeton. edu/node (2014)