Survey of Approaches and Techniques for Security Verification of Computer Systems

FERHAT ERATA*, Yale University, USA
SHUWEN DENG*, Yale University, USA
FAISAL ZAGHLOUL*, Yale University, USA
WENJIE XIONG*, Virginia Tech, USA
ONUR DEMIR†, Yeditepe Üniversitesi, Turkey
JAKUB SZEFER*, Yale University, USA

This paper surveys the landscape of security verification approaches and techniques for computer systems at various 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 not sufficient to verify just the software levels or just the hardware levels in a mutually exclusive fashion. This survey especially highlights system levels that are verified by the different existing projects and presents to the readers the state of the art in hardware and software system security verification. Few approaches come close to providing full-system verification, and there is still much room for improvement.

CCS Concepts: • **Security and privacy → Formal security models**; *Security in hardware.*

Additional Key Words and Phrases: formal methods, theorem provers, model checkers, security verification, hardware-level verification, software-level verification

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 [86]. Today's computing systems are very complex, and if the design of the hardware, software, or the way the hardware and software interact are not perfect, then there may be security vulnerabilities that attackers can exploit. To help find these potential vulnerabilities and prove the designed system is trustworthy, formal methods can be used. For instance, in the development of eXecute Only Memory (XOM) computer system [68] formal verification was used and a possible replay attack was identified, and then the design was improved and proved to be secure. Since the security of systems such as XOM depends on the correctness of the protections that both the hardware and software components provide, there is a need to verify the security of both the hardware and the software before commercializing the system.

Unlike secure architectures such as XOM, most computer architectures and systems are not formally verified. This shortcoming leaves them open to potential vulnerabilities and future security attacks. To help protect the systems, and to promote more security verification of computer systems, this survey aims to show readers about existing approaches to security verification of computer systems. In this survey, we compare different projects and tools that consider both the hardware and the software levels of a system, and which use formal methods to verify security properties of such systems. The different approaches and tools are analyzed and discussed in detail to help

*This work was supported in part by the National Science Foundation (NSF) grants 1419869 and 1524680; and Semiconductor Research Corporation (SRC) contract 2015-TS-2633. Shuwen Deng was supported through Google PhD Fellowship.
†The author's work is supported by TUBITAK grant 2219.

Authors' addresses: Ferhat Erata, Yale University, USA, ; Shuwen Deng, Yale University, USA, ; Faisal Zaghloul, Yale University, USA, ; Wenjie Xiong, Virginia Tech, USA, ; Onur Demir, Yeditepe Üniversitesi, Turkey, ; Jakub Szefer, Yale University, USA, .

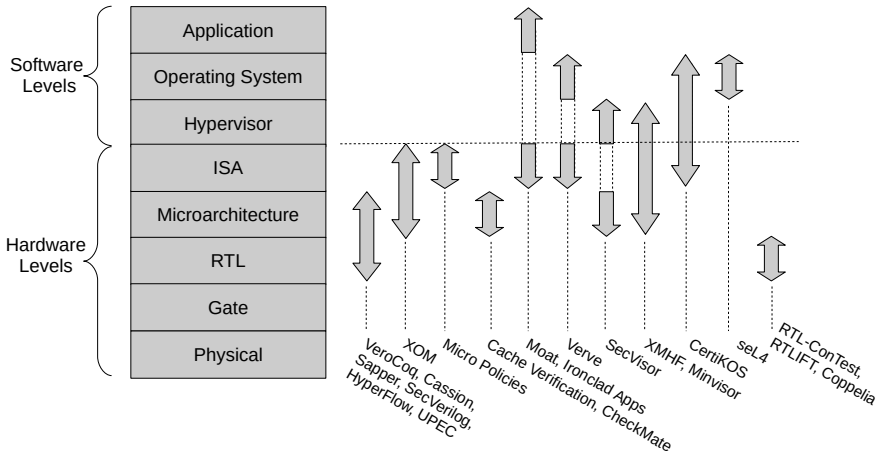

Fig. 1. Hardware and software levels found in a typical computer system are shown on the left. The different security verification projects and levels which they consider are shown on the right. The verification projects surveyed in this work focus typically on one or more of the levels. Broadly, projects either focus on multiple software levels, or hardware levels; some projects span both software and hardware levels, but not all the levels. Note, some projects "skip" certain levels, as indicated by the breaks in the arrows on the right-hand side of the figure.

readers understand the state of the art in security verification and lower the barrier to enter into this research field for interested researchers.

1.1 Software and Hardware System Levels Considered in the Verification Process

A computer system is typically composed of multiple hardware and software levels, as shown in Figure 1. The typical software levels are Application, Operating System (OS), and Hypervisor. These levels cover typical software running on a commodity computing system. The common hardware levels in a computer system 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. E.g., a guest OS relies on the Hypervisor to provide isolation from other malicious guests, if the more privileged Hypervisor has a security vulnerability, the OS cannot make any guarantees about security. For example, an ISA is not secure at the hardware level if the microarchitecture that implements it has a bug. Likewise, a microarchitecture realized using a flawed RTL is not secure, and so forth. The relationship is not strictly linear, in that the upper level always depends on all lower levels. Some secure architectures have introduced hardware that allow higher software levels to be protected from intermediate software levels. For example, in Bastion [20] applications are able to communicate with the Hypervisor while bypassing the OS; or in HyperWall [96] a virtual machine does not need to rely on hypervisor for isolation as the hardware provides some basic memory management functionality.

The software and hardware levels needed to ensure security of the system constitute the trusted computing base (TCB) which contains all the software and hardware that need to be trusted (although it may not be trustworthy). Thus, the TCB should be verified for security to make it truly trustworthy. Effectively, TCB consists of different components at different levels, and security verification tools and methods should include all the levels in the TCB when checking the security of the system. Figure 1 on the right side shows the various projects surveyed in this work and the

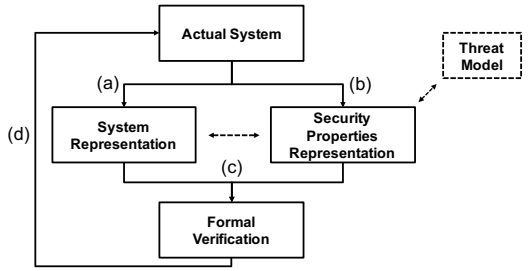

Fig. 2. General procedure for security verification of a computer system.

different system levels that their security verification covers. Because different projects consider different levels, it may be difficult to select the right approach (or mix of approaches) for security verification that one may desire. Especially, some of the works skip certain levels, which may not be needed for their verification, for example Moat [94] verifies applications with respect to ISA, and assumes hardware fully protects the applications from OS or Hypervisor, so OS and Hypervisor levels are skipped. By studying each group of projects, this survey aims to show the state of the art in security verification of hardware and software systems and allow researchers and practitioners to understand how to approach security verification of their designs competently.

2 TOOLS AND MECHANISMS USED IN SECURITY VERIFICATION

This section presents a background on the different tools, mechanisms, and approaches typically used to check security guarantees of a computer system. 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.

2.1 System Representation

In order to verify if a system complies with some properties, we need a representation of the system that accurately expresses the behavior of the system. 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 separate 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 [97] and VHDL [70]. 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) [72], including Chisel [6], BlueSpec [79],

and Genesis2 [90]. Most recently, there is active deployment of High-Level Synthesis (HLS) where the hardware is inferred from software-like description of the algorithm or a program that it should realize. There are some attempts to link these hardware and system representation methods with security property specification to realize their security verification, which we will present in Section 3.

Software components, meanwhile, can be described by their high-level implementation in programming languages such as C, C++, or Java. There are ongoing efforts to integrate design processes and system specifications with security-property specifications and include verification information inside programming languages as annotations. Examples include TAL [28] and Dafny [61], which we will present in Section 4.

Alternatively, some tools are developed to generate a model in the verification tool from its original description in HDL or in software programming language. 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 an automated method of creating a model is not available, then the 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. These are also described in Section 3.

Formal verification is done with respect to a system representation, as described above. Most projects assume a trusted compiler or toolchain such that the generated system implementation flawlessly matches the system representation and does not contain extra hidden or unwanted functionality that may compromise the system's security. 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 [98], where a virus-infected compiler was 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 [62] which is a certified compiler that generates binaries from Coq code. All surveyed work assumes trusted toolchains.

2.2 Representation of Security Properties

Depending on the verification mechanism, either *deductive mechanisms* or *algorithmic mechanisms* can be used, as detailed in Section 2.3 and 2.4. In deductive mechanisms, the security properties can be represented in terms of *logical formulas*. A logical formula serves as a limitation 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.

2.3 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.

Theorem provers (a.k.a. proof assistants) aid the verification process by providing frameworks for creating a mathematical model of the system, for specifying the security properties, and 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 are many proof assistants used actively in academia and industry, such as Coq [12],

Isabelle/HOL [80], PVS (Prototype Verification System) [82], ACL2 (A Computational Logic for Applicative Common Lisp) [56], and Twelf (LF) [47]. 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.

2.4 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, SMT (Satisfiability Modulo Theories) Solvers, and Symbolic Execution fall in this category.

As defined in [7], "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 is determined by the invariants and the complexity of the model. The output can be positive (property satisfied), negative (property violated), or the execution runs indefinitely. There are a number of model checking tools: SPIN [51], Murφ [33], SMV [74], CBMC [59] For further details about model-checker design, we point the readers to an early survey by Clarke et al. [23]. Model checking has a well-known *state explosion problem* [24, 103], 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 simplify the model. However, as the level of abstraction gets higher, we run the risk of missing some important details of the system design.

Satisfiability Modulo Theories (SMT) solvers are used to solve satisfiability problems expressed in first-order logic over arithmetic, bit-vectors, string and arrays. They rely on efficient satisfiability procedures for propositional logic (SAT solvers). SMT solvers such as Z3 [31], CVC5 [9], and MathSAT5 [22] have been used as a building block for a wide range of applications from program analysis to program verification and software testing. For verifying systems, first, the system representation and security properties have to be transferred into formulas that SMT solvers can work with. The validity of the property is then checked by SMT solvers [11]. Note that SMT solver execution time can vary from a few seconds to hours depending on the size of the problem. There are many verification tools using SMT solvers. For instance, Z3 [31] is an SMT solver used by an intermediate verification language, Boogie [108], and Dafny [61] is a programming language and verifier for functional correctness that uses Boogie as its target language. Vale [16], a language for expressing and verifying high-performance assembly code, uses Dafny/Z3 as its verifier backend.

Alloy [53] is a formal specification language based on first-order relational logic. Alloy's reasoning engine uses SAT solving technology to simulate designs and find subtle flaws, and has been used in a wide variety of applications from networking and security to critical systems [54].

Unlike model checking, which requires a model of the system, symbolic execution [8] deals directly with the program semantics assuming symbolic values for inputs. It thus arrives at expressions in terms of those symbols and constraints them with the possible outcomes of each conditional branch. Finally, program expressions can be evaluated by solving the constraints (e.g., by an SMT-Solver). In this way, all possible execution states can be evaluated simultaneously, at great cost of storage and slow execution during analysis. There are many symbolic execution engines targeting different software levels: while KLEE [18] symbolically executes programs in the LLVM Intermediate Representation (IR) [60], Angr [91] executes lifted binary programs in VEX IR [78], and JPF [49] executes Java bytecode. Apart from that, there are also tools such as

UCLID5 [89] created to meet the requirements from different aspects, including providing a natural way to model both concurrent transition systems and sequential software with expressive abstract datatypes, specifying a range of properties, providing a diverse palette of verification methods supported by state-of-the-art computational reasoning engines including those for synthesis and learning, supporting compositional reasoning, and giving counterexamples and feedback to users.

3 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. As discussed in Section 2, the formal verification tools have their own languages with formal semantics for describing system specification, security properties or invariants, and for doing the verification and proofs. However, these languages differ significantly from common HDLs (Hardware Description Languages) that are used in hardware design.

To bridge the gap, in order to perform security verification, researchers most often either manually or automatically convert system designs in HDL to system representations in formal verification tools' languages; or, conversely, generate HDL specification from the system representation written in a verification tool's language. With the former approach, hardware designers can design the system in HDL as usual, and the main new effort is in generating models in the verification tools' language and describing the security specification. With the latter approach, hardware designers need to learn the language used by the verification tools and develop hardware, as well as the security specification and proofs with that language. The tools then generate HDLs from the system representation, so the system can be synthesized normally with the existing toolchain. It is required to manually annotate the whole codebase for security verification and know beforehand which sources and sinks for the information flow.

The following classes of projects are discussed next: manually modeling systems in verification tools' languages, automatically converting system designs in HDLs to models in verification tool's languages, adding verification features to existing HDLs, or generating HDLs from system models written in verification tool's language.

3.1 Approaches Requiring Manually Modeling Systems in Verification Tools' Languages

Since the languages used in verification tools are usually very different from HDL, for security-critical modules, sometimes it makes sense to manually model the system and verify the model with security specifications. Depending on the size of the system, this process is time-consuming. Moreover, this process does not guarantee that the model faithfully represents the real system.

If a *proof assistant* is used, the system is modeled as a set of definitions, and the security properties are formalized as theorems. Then the proof is developed manually and checked by the proof assistant. If a *model checker* is used, the model of the system is built in the form of a finite state machine (FSM). 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. Usually, the model is simplified to avoid the state explosion problem.

3.1.1 Micro-policies: verifying PUMP secure architecture. The work on Programmable Unit for Metadata Processing (PUMP) [32] added a programmable metadata processing unit alongside with the data computation. PUMP allows programmers to create policies and rules that enforce IFT (Information Flow Tracking) mechanisms by manipulating the metadata tags in each instruction. Metadata processing can thus support many safety and security policies. However, given a high-level specification, it is nontrivial to design metadata processing rules. Whether the metadata processing rules in PUMP comply with high-level security properties needs to be proved. Micro-Policies [5, 30] presented an approach for formalizing and verifying the IFT policies.

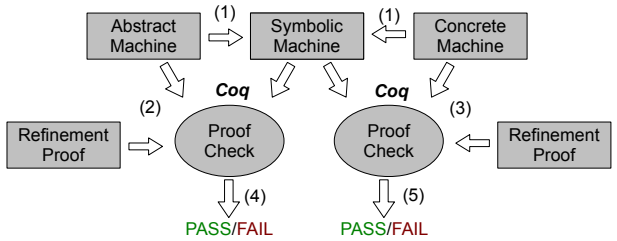

Fig. 3. Micro-Policies verification process.

Micro-Policies verification process is shown in Figure 3. To design a set of metadata rules, first, an abstract machine specification with a set of instructions and information flow policies is defined, showing the 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 about whether the concrete machine reflects the abstract machine specification, an intermediate layer *symbolic machine* is added manually, as shown by arrows labeled (1) in Figure 3. The Micro-Policies prove the equivalence by backward refinement, which means if there is a state transition in a low-level machine, there exists a corresponding transition in a high-level machine. They use Coq to formally prove whether the concrete machine backward refines the symbolic machine (2), and whether the symbolic machine backward refines the abstract machine (3). If the backward refinement verification in both (4) and (5) passes, then the concrete machine has the security properties of the abstract machines.

The work shows the proof of a variety of security policies, including noninterference, sealing, compartmentalization, control flow integrity, and memory safety. The whole verification process requires about 17.7k lines of code. To apply this methodology to other architectures, abstract, concrete, and symbolic machines need to be specified by the designer manually for each architecture. The refinement proofs depend on the system and also need to be re-done. Currently, there is no programmatic way to generate these from the HDL code. Reusability of this approach is low.

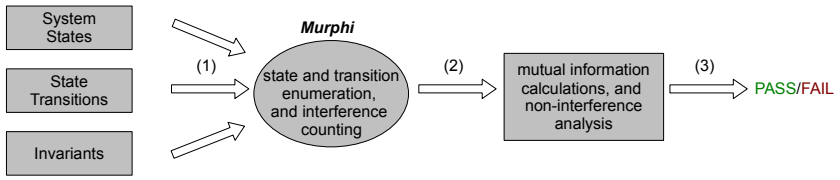

Fig. 4. Processor cache security verification process using Murφ model checker.

3.1.2 Cache Verification with Murφ. Processor caches are an 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 no 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., [71], that are able to compromise data confidentiality and potentially leak out cryptographic keys.

In [113] researchers create side-channel leakage models based on the non-interference property between an attacker and a victim process that are using the same processor cache. First, they model

the cache architecture in Murφ 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.), (1) in Figure 4. 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. Zhang and Lee used Murφ to enumerate all possible states and transitions, and count the number of interferences between attacker and victim for the different state transitions, (2) in Figure 4. Based on this data, mutual information [27] is then used to quantitatively analyze the interference between the two processes, and reveal side-channel vulnerabilities, (3) in Figure 4.

Authors of [113] applied their work to six cache architectures and revealed that most cache architectures do not satisfy the non-interference property, thus failing 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). Reusability of this approach is low.

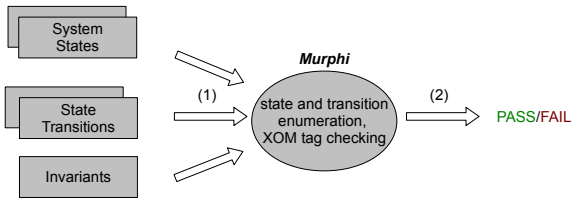

Fig. 5. Process for verifying XOM architecture with Murφ model checker. Two sets of states and state transitions are show, corresponding to the “actual” and “ideal” worlds that XOM verification process compares during verification.

3.1.3 Verifying XOM Architecture. The eXecute Only Memory [68] 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 which label the identity of the owner of the data, while off-chip data is protected by encryption and hashing. In [67], XOM was formally specified and then verified in Murφ.

A model of XOM and its adversary is built in Murφ as shown in Figure 5. The model of the XOM hardware contains arrays representing the registers, cache, and memory including data and tags. The possible values and states the hardware is modeled as system states, (1) in Figure 5. The effects of each operation of the processor are represented as state transitions. 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 transitions. 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, (2) in Figure 5: to prove the adversary cannot read user data, the model checker verifies that the on-chip user data is tagged with user’s XOM ID and off-chip user data is always encrypted and hashed with the user’s key. To prove that the adversary is not able to write the user data without halting the system, the model checker compares the state of the ideal world against the state of the actual world, and thus, knows whether the adversary will succeed.

The authors of XOM, during verification, found a replay attack and fixed it. Moreover, it was shown that if the operating system does not behave maliciously the liveness of the system is guaranteed. To apply this method to other designs, designers need to manually create the Murφ system representation for the architecture. Especially, invariants about any tags need to be specified. Again, there is currently no automated way to extract system models from the HDL system representation. Reusability of this approach is low.

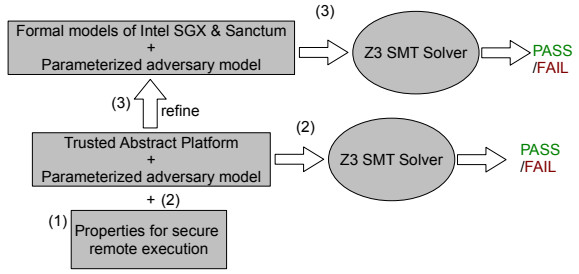

Fig. 6. Process for verifying formal models of trusted hardware platform like Intel SGX and Sanctum

3.1.4 Formal Foundation of Enclave Secure Remote Execution. Enclave is a special kind of CPU that is able to maintain a protected memory region and take advantage of this to do operation or isolation of sensitive code and data. The formal foundation for Secure Remote Execution (SRE) of Enclaves [95] provides a framework and methodology to reason about the security guarantees provided by enclave platforms.

As is shown in Figure 6, first, secure remote execution of the enclave is defined. It is used to provide formal security features to let users remotely outsource the enclave's execution including attestation and secure operations of data. Three key security properties that entail SRE are integrity, confidentiality, and secure measurement. Secure measurement allows the user to verify that the platform is running unmodified enclave programs. The secure measurement property states that any two enclaves with the same measurement must also have the same semantics: they must produce equivalent execution traces for equivalent input sequences. Secondly, a Trusted Abstract Platform (TAP) is introduced to specify trusted primitives of enclaves' behavior. Along with that a parameterized attacker model is defined to verify TAP's confidentiality, etc. Proof is provided that secure remote execution holds for TAP. In the final step, the ideal TAP is refined, and the refined platform is shown to have equivalent functionality and security compared with some practical enclaves like Intel SGX [25] and MIT Sanctum [26]. Consequently, these practical trusted hardware platforms are verified to hold SRE.

All the hardware platform models including TAP, Intel SGX and MIT Sanctum are constructed by BoogiePL and verified by Z3 SMT solver. This formal foundation is proved to be able to efficiently and effectively verify SRE of enclaves.

3.1.5 CheckMate. CheckMate [100, 101] is an approach and automated tool for determining whether a microarchitecture design is vulnerable to a given class of security vulnerabilities. It can also be used to automatically synthesize proof-of-concept exploit code for real-world exploits. Figure 7 shows the overview of the CheckMate toolflow.

CheckMate requires two inputs: manual microarchitecture specification (μspec model) and specification of a class of exploits (exploit pattern). First, a microarchitecture specification, which includes an axiomatic description of a hardware design and its related OS support, defines microarchitectural structures that micro-ops pass through at various points of execution, hardware-supported

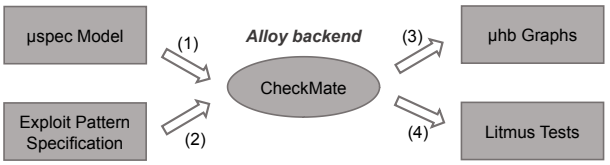

Fig. 7. Overview of the CheckMate toolflow.

micro-ops, and any hardware-specific execution event orderings (e.g., in-order Fetch or out-of-order execution). Recently, [52] presented a tool that takes a microarchitecture design in Verilog or systemVerilog and automatically converts RTL into μspec. The CheckMate will parse the microarchitecture specification and translate it into a relational model. After that, Alloy [54], which is utilized to implement CheckMate, maps the model into SAT. Then, CheckMate will synthesize the candidate executions and utilize the second input, exploit pattern specification, which is the formalization of hardware execution patterns indicative of security exploit classes, to do implementation-aware candidate pruning. This will help output the μhb graphs representative of hardware-specific exploit program executions when the input microarchitecture is detected to be vulnerable to the input vulnerability. Finally, the security litmus tests can be extracted. Security litmus tests are the most compact representation of an exploit program. Security litmus tests are much more practical to analyze with formal techniques than a full program due to their compact nature. On the other hand, they are easily transformed into full executable programs if needed.

CheckMate is able to showcase real-world hardware security vulnerability detection. It is used to supply a speculative out-of-order (OoO) processor and a FLUSH+RELOAD [109] cache side-channel attack exploit pattern. From these inputs, CheckMate is able to synthesize programs representative of Meltdown [69] and Spectre [58] attacks. Next, holding the microarchitecture constant and replacing the FLUSH+RELOAD exploit pattern with a PRIME+PROBE [81] exploit pattern. CheckMate further generates corresponding new attacks and is shown to be able to leak private information. This result further validates the CheckMate approach to automated synthesis of real-world exploits.

3.2 Automatic Conversion from HDL to System Models in Verification Tools

To lower the verification efforts, there are attempts to develop tools that automatically convert designs in HDL to system models that are used in verification tools. However, the security specification and proofs still need to be done as a verification effort. For certain security properties, it is possible to automatically generate security specifications and proofs.

3.2.1 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 to verify the security of the design written in Verilog [14]. Original VeriCoq supports an essential subset of Verilog, but requires the design to be flattened and have no nested modules. The newer VeriCoq-IFT [13] has the same constraints, but adds the ability to verify information flow properties automatically. The information flow proofs need "initial sensitivity list" as input and labeling the variables in the design. Given this input, VeriCoq-IFT automatically creates theorems and proofs for guaranteeing the information flow property.

The verification process is shown in Figure 8. First, the input is the Verilog code, which is then converted into Coq by VeriCoq, (1) in Figure 8. Based on the security properties requested, designers create the theorems to be verified, (2) in Figure 8. With the design represented in Coq, alongside with the theorems, developers come up with formal proofs showing that the code has the desired

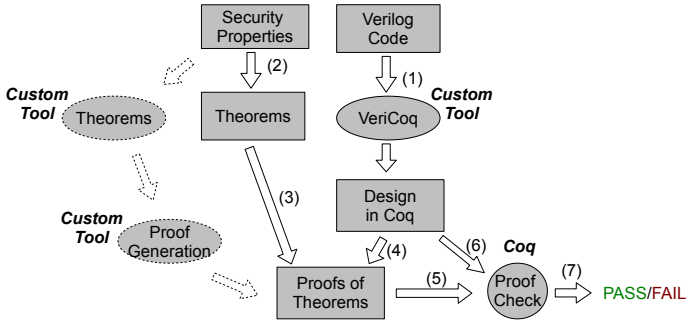

Fig. 8. VeriCoq and VeriCoq-IFT verification process. Dashed portion on left-hand side shows the VeriCoq-IFT process that can automatically generate the theorems and the proofs.

security properties, (3) and (4) in Figure 8. They can then verify that the design aligns with the defined security properties by using Coq, (5) and (6) in Figure 8, and either the design passes, meaning that it conforms the properties, or it fails to pass, (7) in Figure 8.

The advantage of VeriCoq is the automated conversion of Verilog code into Coq. VeriCoq-IFT also adds the ability to automatically generate the theorems and proofs for information flow. To apply this method to other designs, security properties need to be specified, and the theorems and proofs to be developed manually since they are not focusing on information flow. Reusability of this approach is medium.

3.2.2 Formal-HDL. Formal-HDL [55] is a hardware description language in Coq proof assistant. In [46], a tool is developed to automatically convert design in VHDL to *Formal-HDL*. Different from VeriCoq, which only allows a flattened hierarchical design (a one-level design), *Formal-HDL* supports instantiation of modules within other modules.

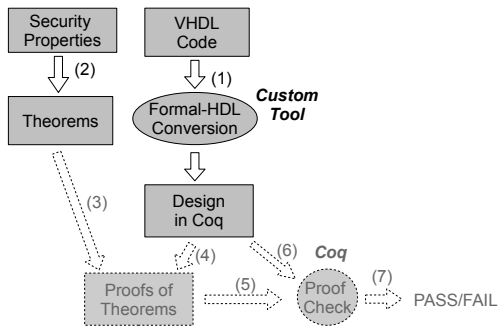

Fig. 9. Formal-HDL verification process, the lightly shaded portions are not yet done and are presumably future work of the authors.

The advantage of Formal-HDL is the automated conversion of VHDL code into Coq. Also, it supports instantiation of modules within other modules. However, currently no actual security verification is done using the Coq model. To apply this method to other designs, VHDL can be automatically translated to Coq, but all security verification work has to be done manually. Reusability of this approach is low (as the current work [55] does not do any actual proofs about security, just produces Coq model).

3.2.3 UPEC. Unique Program Execution Checking (UPEC) [34, 35] is a formal method that is used to detect and locate vulnerabilities to covert channels systematically, including unknown covert channels. A new covert channel was found using UPEC in the RISC-V Rocketchip [4]. There is also an example of ISA non-compliance found by the UPEC in the implementation of the RISC-V Physical Memory Protection (PMP) mechanism in Rocketchip.

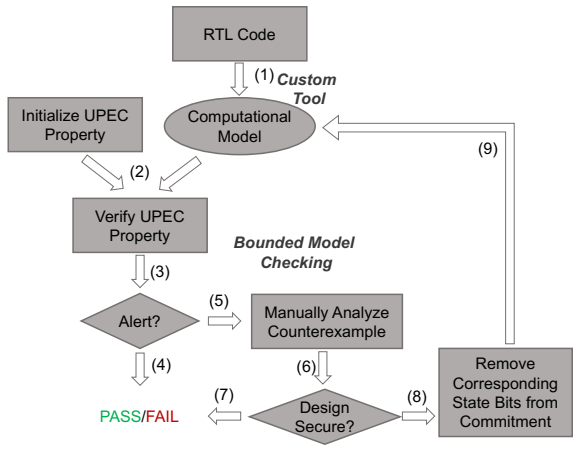

Fig. 10. Unique Program Execution Checking(UPEC) verification process.

The verification flow, as is shown in Figure 10, is similar to Micro-Policies [32], shown in Section 3.1.1, but targets on covert-channel detection similar to CheckMate [100], shown in Section 3.1.5. The difference is that the computational model that is used for the unique program execution checking can be derived automatically from the RTL description of the design and the user only needs to specify the protected memory region which represents the memory region holding the secret data.

3.2.4 RTL-ConTest. RTL-ConTest [75] is a comprehensive framework for efficiently validating the security properties and detecting security vulnerabilities of a System-on-Chip (SoC). It performs RTL-level concolic testing, which combines symbolic execution with concrete simulation, to detect security vulnerabilities manifested in the hardware design of the SoC.

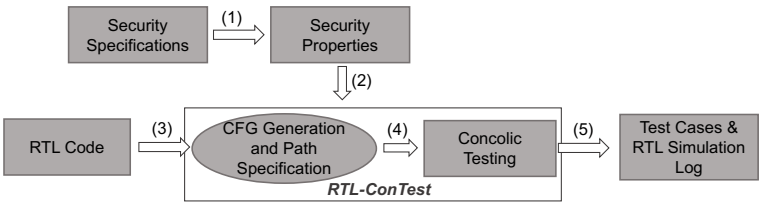

Fig. 11. RTL-ConTest framework.

As is shown in Figure 11, RTL-ConTest first extracts the process flow for symbolic execution by generating the critical flows of target RTL as control flow graphs (CFGs). This CFG generator drives the concrete path specification and concolic testing algorithms in order to generate RTL-level security test cases. Furthermore, the test cases are validated by the preestablished security properties to detect critical violations in the SoC RTL. Similarly to UPEC [35], the proposed RTL-ConTest

algorithm is also evaluated on two open-source RISC-V-based SoCs and is able to find vulnerabilities both within the core and in the rest of SoC, e.g., incorrect password checking logic in the debug unit, or the address range overlap between the peripherals master and SoC.

3.2.5 RTLIFT. Information Flow Tracking (IFT) has been widely used to enforce security properties, such as confidentiality, integrity, and non-interference [85]. To precisely reason about the security properties in a hardware design, RTLIFT [3] tracks information flow at the RTL level. The RTLIFT software generates extra IFT logic in the circuit and then, with standard functional verification tools, it can evaluate the security property (information flow) of the hardware design, i.e., make sure no *High* data flows to *Low* outputs. After the verification, the extra IFT logic is removed from the design – so the verification does not introduce overhead into the final circuit design.

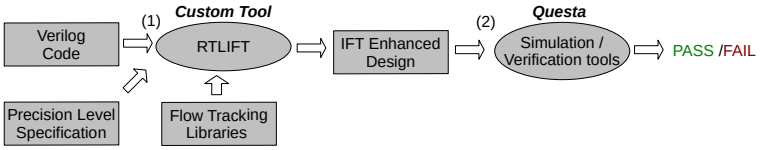

Fig. 12. Flow of the RTLIFT verification process.

To generate IFT logic automatically, flow tracking libraries were developed for Verilog for each basic module such as a multiplexer, decoder, etc. For each basic module two types of information flow tracking were considered, along with an associated library: a precise library and a conservative library. The precise library propagates the security tags of signals in such a way as to minimize the number of false positives, while the conservative library gives smaller tracking logic with simple OR expression, but may generate more false positives. As shown in Figure 12 (1), Verilog code, along with the flow tracking libraries and specification of the desired precision (precise or conservative), is used by RTLIFT to replace each basic module in the original Verilog code with its corresponding module from the library that allows for tracking of flows through that basic module. To deal with implicit information flows in the hardware design code, conditional statements are treated as explicit multiplexers where security tags of multiplexer control signals propagate to the multiplexer output. Next, as shown in Figure 12 (2), The generated circuit with IFT features is then fed to simulation and verification tools (Quetsa Formal Verification tool in this case) to analyze whether unwanted information flows exist. If the circuit passes the verification, then the extra IFT checking logic that was added can be removed from the hardware design, while the design maintains its security properties.

Compared to tracking the IFT at the gate level [99], RTLIFT has more information about the high-level circuit, and thus, can propagate tags faster and more precisely. An RSA core, an AES core and a bus architecture were verified using this method and hardware Trojans in the designs, which leaked secret keys to output, were detected. To apply this method to other designs, the information flow libraries can be re-used. Reusability of this approach is medium-high (slightly more reusable than others so far as design specific theorems or libraries need not be developed if only *High* to *Low* IFT is considered).

3.2.6 Coppelia. Coppelia [112] is an end-to-end automated exploit generation approach to validate security properties of processor designs. Given a processor design and a set of security properties, Coppelia systematically explores the design using symbolic execution, and if security violations are found it generates C programs with inline assembly that exploit bugs within the design. It adapts KLEE [18] for the symbolic exploration of hardware designs at the register transfer level (RTL).

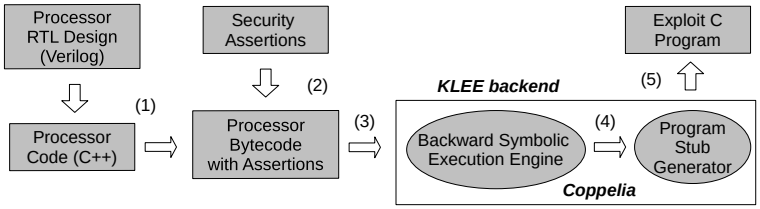

Fig. 13. Coppelia's tooling workflow.

As shown in Figure 13, Coppelia takes as input an HDL implementation of a hardware design and a set of security-critical assertions. The RTL design in Verilog is transformed into C++ (1), and after translation, Coppelia adds the security-critical assertions (2) and compiles the newly translated design to LLVM bytecode [60] (3) using the Clang compiler. The reason for LLVM bytecode generation is that Coppelia is built on top of KLEE which is an LLVM-based symbolic execution engine. Coppelia then uses the generated C programs to simulate the design and check whether the design is vulnerable to the security properties using backward symbolic exploration. Once a violating state is detected, Coppelia builds the sequence backwards all the way to the initial state and finds inputs (4). Using these inputs it generates an exploit by adding program stubs in C (5). These program stubs are generated according to the category of the security-critical assertion violated such as control-flow, exception, and memory-access related. Coppelia is evaluated on three processor designs: OR1200, PULPino, and Mor1kx, representing two different RISC architectures: OR1k and RISC-V.

3.3 Adding Security Verification Features in HDL

Another approach for security verification is to add security verification features into an HDL, e.g., by either introducing a new HDL language or introducing new syntax into existing language. Caisson [65], Sapper [64], and SecVerilog [111] take this approach and introduce information flow tracking (IFT) features into an HDL language. System designers can use new syntax to specify the information flow tags and policies in their designs. If the verification passes, then designers know their designs do not have any information flow violations.

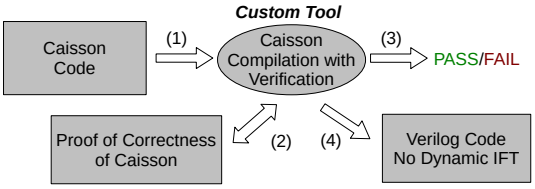

Fig. 14. Static information flow verification process of Caisson.

3.3.1 Caisson. Caisson [65] is a hardware description language for static information flow verification at the design time. The verification process is shown in Figure 14. First, the original Caisson code is written containing labeling of security tags, especially data ports in a hardware module are assigned with security labels (i.e. *Low* to *High*), (1) in Figure 14. With the design written in Caisson, and with the security labels on each register and wires, the information flow can be checked at the design time by running the Caisson compiler, (3) in Figure 14. During compilation, it is checked whether the information flow strictly follows the policy that data labeled *High* should not end up

in a port labeled *Low*. If so, there will be no information flow form *High* to *Low* during the system runtime. The checking is done based on the typed Caisson language and type checking rules in the Caisson tools. Caisson can generate standard Verilog code as the output as well, with no labels, and the code can be synthesized using existing tools, (4) in Figure 14, i.e. labels are removed and have no impact on final design or performance. In the paper, authors use manual proofs to formally prove that Caisson enforces timing-sensitive non-interference in designed hardware, (2) in Figure 14.

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 [65]. To apply this method to other designs, the designer needs to augment his or her Verilog code with the security labels. Reusability of this approach is medium.

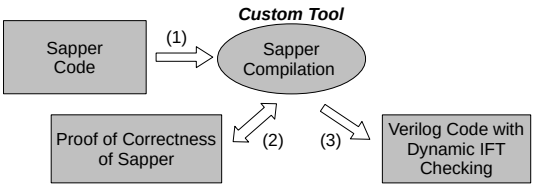

Fig. 15. Dynamic information flow verification process in Sapper.

3.3.2 Sapper. Sapper [64] is a hardware description language that is based on a synthesizable subset of Verilog. Sapper compiler automatically ensures non-interference in the generated hardware logic, and is able to generate Verilog code with added dynamic information flow tags. Figure 15 shows the verification flow.

First, the Sapper code is written, which includes labeling of security tags and in particular input and output ports in a hardware modules are assigned with security labels (i.e. *Low* to *High*), (1) in Figure 15. Example of how security labels indicating an IFT policy are inserted into the code in Sapper language shown in Figure 16. Sapper's policy is that the hardware logic should ensure that data flow to any output port never allows *High* data to reach a *Low* port. Especially, in the presence of an active attacker (e.g., a malicious software in the system), who has full control over all *Low* input ports, the non-interference enforced by the policy can protect all the data tagged with *High*. Sapper statically analyzes the hardware logic and automatically inserts dynamic IFT logic and generates Verilog code with extra logic for the dynamic information flow tracking, (3) in Figure 15. In the paper, authors use pen-and-paper proofs to formally prove that Sapper enforces non-interference of the generated system, (2) in Figure 15.

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. Authors designed a processor simulating the hardware with ModelSim [48]. A micro-kernel and a compiler were also implemented, and processes in different security levels could run on the processor. To apply this method to other designs, the designer needs to write his or her Sapper code with the security labels. Reusability of this approach is medium.

3.3.3 SecVerilog. SecVerilog [111] is a well-typed language and built on top of Verilog to include information flow annotations. It is first proposed in [110] to mitigate the timing channel in program execution. The language semantics can be used to analyze and formally prove the security of the system.

SecVerilog enables static checking of hardware information flows and uses an SMT checker to verify non-interference between modules with different security levels. First, the designers define

	Sapper	Verilog

check	`reg[7:0] a: %*\hl{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. 16. Example Sapper code and generated Verilog code, modeled after [64], with a security label highlighted.

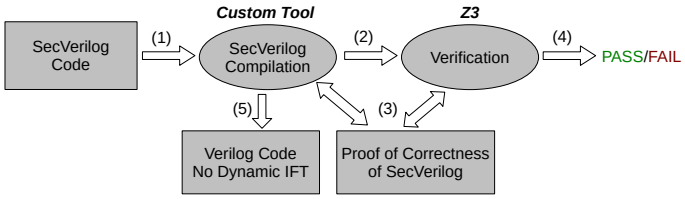

Fig. 17. Static information flow verification process of SecVerilog.

a security policy, for example, the design has two security levels: *Low* and *High*. The policy may be such that the adversary, who has access to all information at or below the *Low* security level, and can measure the clock cycles of hardware operations, never has access to any data labeled *High*. Also, during the implementation of the design in SecVerilog, each variable has to be labeled with its corresponding security label, (1) in Figure 17. Example of SecVerilog labeling the code is given in Figure 18. Using these labels, SecVerilog generates models in Z3 for verification, (2) in Figure 17. Then, in Z3, the information flow is checked, and a report is given whether the design passes or fails the verification, (4) in Figure 17. On the other hand, SecVerilog generates designs in Verilog, (5) in Figure 17. In the paper, authors use pen-and-paper proofs to prove SecVerilog enforces timing-sensitive noninterference, (3) in Figure 17.

```
1  reg[18:0] %*\hl{\{L\}}*) tag0[256], tag1[256];
2  reg[18:0] %*\hl{\{H\}}*) tag2[256], tag3[256];
3  wire[7:0] %*\hl{\{L\}}*) index;
4  //Par(0)=Par(1)=L Par(2)=Par(3)=H
5  wire[1:0] %*\hl{\{Par(way)\}}*) way;
6  wire[18:0] %*\hl{\{Par(way)\}}*) tag_in;
7  wire %*\hl{\{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. 18. Example of the split cache in SecVerilog, modeled after [111], with the security labels highlighted.

SecVerilog allows sharing of resources within a module. Static labeling does not solve all the problems of information flow, especially if resources are shared. In the case of shared resources, the labels might change during runtime. SecVerilog uses dependent types to handle runtime label

changes. A design of split cache is shown in Figure 18 as an example. Type changes are detected and updated dynamically during the runtime, e.g., $Par(way)$ in Figure 18. 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 18.

SecVerilogLC [37] extends SecVerilog [111] to allow more sufficient hardware resource-sharing for different security levels. For dependent labels, the information flow control type system along with syntax and semantics supports signals to propagate on clock edges explicitly. In order to test the next clock cycles' label, there are also related syntax supported. The type system permits registers to update values with labels securely and statically. Furthermore, SecVerilogLC explicitly divides sequential and combinational variables to do corresponding security checks. Following the changes illustrated above, it is also able to avoid implicit downgrading by explicitly implementing it.

SecVerilogBL [38] [39] also extends SecVerilog [111], to support packed data structures, and downgrading mechanisms. It provides an improved type system to cover the extensions. The first new feature allows complex data structures such as arrays, network packets to be tagged with finer granularity. That allows tagging of individual elements within arrays or packed data structures. The second feature supports modifying the security tag of an element dynamically.

ChiselFlow [40] integrates another HDL embedded in Scala – Chisel – with the security type system and further extends SecVerilog with new features including nonmalleable downgrades and type inference.

A secure MIPS processor and caches were designed in SecVerilog[111] and SecVerilogLC [37]. Dynamic labeling makes the shared ports of the cache possible. SecVerilogBL is used to verify a secure architecture based on ARM TrustZone which provides isolated memory regions for providing confidentiality and integrity [38] [39]. SecVerilog also provides timing-sensitive non-interference, which is proved in the paper [111]. To apply this method to other designs, the designer needs to write his or her Verilog code with the annotations. Reusability of this approach is medium. Apart from that, a full-featured processor [40] offering a complete RISC-V instruction set prototyped the HyperFlow, with moderate overhead added to the area and the performance.

In [110], a well-typed language is proposed to mitigate the timing channel in program execution. Each command in the program is extended with security labels for confidentiality and integrity, and a new command "mitigate" is introduced to bound the execution time of another command. The language made some assumptions on the properties of the underlying hardware. The language semantics can be used to analyze and formally prove the security of the system. Meanwhile, a secure hardware architecture satisfying the properties required by the language is designed, explicitly formalized and experimentally shown to have only moderate overhead.

3.4 Generate HDL from System Model in Verification Tools

Another approach is to develop a new domain specific language, model and verify the system using this domain specific language (and associated tools), and then generate HDL. The hardware designers need to learn and use the new language, but the tools will then automatically generate HDLs, so there is a one-to-one relationship between the code used for verification and the final HDL code.

3.4.1 ReWire. ReWire [84] is a functional programming language and compiler that translates high-level designs into HDL description of the hardware. It is a subset of Haskell, which produces a suitable foundation for writing formal specifications. ReWire enables modular, high-level, semantics-directed hardware circuit designs.

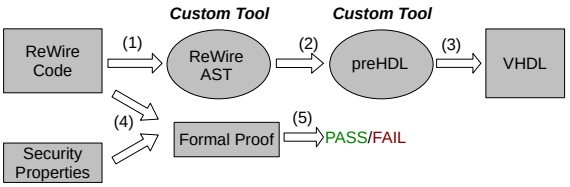

Fig. 19. ReWire verification flow

In ReWire, combinational logic is represented by pure, non-recursive first-order functions. The sequential logic in one clock domain is represented by a structure called "reactive resumption monad." This structure uses tail recursive type and functions as a continuation to map an input to a "new" sequential logic. Monad is the method to produce new types with functions of specific computation by robustly incorporating fundamental data type values. More information on monads can be found in [66]. As shown in Figure 19 (1), to generate synthesizable VHDL code from ReWire code, first the ReWire abstract syntax tree (AST) is produced by parsing Haskell concrete syntax. (2) A preHDL is generated by compiling the monadic operations and loop flattening. (3) By replacing the loop structures with VHDL processes, preHDL can be converted to VHDL.

In a sample dual-core processor with shared register, one core is designated as *High* core and the other is *Low*. In ReWire, to verify the separation between the two cores as theorems, a formal proof is written in Haskell, as in Figure 19 (4). The verification precludes storage channels, timing channels and control flow channels. In the proof, a "harness security" function enables precise control of information flow.

With ReWire, a single-core processor and a secure dual-core processor based on the single-core processor were designed and synthesized, showing that ReWire compiler can produce VHDL implementation from the high-level specification and that it supports modular design. To apply this method to other designs, the designer needs to write his or her ReWire code and specify the security properties. Formal proofs have to be done manually but the ReWire AST and preHDL steps are done automatically, so VHDL will be automatically generated from the ReWire code. Reusability of this approach is medium.

3.5 Comparison of Verification Focusing on the Hardware Levels of a System

Caisson and SecVerilog only do compile-time IFT checking. Sapper does both compile-time checking and adds run-time IFT checking into the design, at the cost of extra hardware and decreased performance. Caisson and Sapper do not allow sharing of resources, while SecVerilog adds dynamic labels which allow one module to work on both *High* and *Low* data. The static IFT checking in SecVerilog makes sure that there is no possible combination of operations or inputs that would leak the *High* data to *Low* data – the cost is that without run-time IFT, the design has to be very conservative and considers worst-case scenarios.

Unlike Caisson [65], data with different security labels can share resources in Sapper, e.g., registers, resulting in a lower overhead.

A comparison is shown in Table 1. Caisson and SecVerilog use compile-time information flow control, while Sapper uses run-time information flow control. Compile-time information flow tracking is done at the design time, and does not introduce any run-time 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 [65], dynamic labeling in SecVerilog and run-time IFT in Sapper make resource sharing possible, thereby reducing area and timing overheads.

Table 1. Comparisons of Caisson, SecVerilog and Sapper

	Caisson	SecVerilog	Sapper
Type of Information Flow Checking	Compile-Time	Compile-Time	Run-Time
Type of Label	Static Label	Dynamic Label	Dynamic Label
Prototype Circuit	Time Mux pipeline; Split Cache, Split Ports	Time Mux pipeline; Split Cache, Shared Ports	Time Mux pipeline; Split Cache, Shared Ports

Table 2 compares different verification tools focusing on the hardware levels of a system. Verification to HDL Code Relationship column shows whether there is a programmatic way to verify HDL code or manual effort is needed to construct the corresponding model from HDL for verification. Reusability column shows the effort to verify a new system utilizing the corresponding verification tools as well as the effort needed to verify a new property for the system.

Table 2. Comparison of verification tools focusing on the hardware levels of a system.

	Verif. to HDL Code Relationship	Reusability	Source Code Public
Micro-Policies	manual	low	https://github.com/micro-policies
Cache Verif.	manual	low	no
XOM	manual	low	no
VeriCoq	programmatic	medium	no
Formal-HDL	programmatic	low	no
CheckMate	manual	medium-high	https://github.com/ctrippel/checkmate
UPEC	programmatic	medium-high	https://github.com/tojauch/riscv-boom-UPEC
RTL-ConTest	programmatic	medium-high	no
RTLIFT	programmatic	medium-high	no
Coppelia	programmatic	medium-high	https://github.com/rzhang2285/Coppelia
Caisson	programmatic	medium	https://github.com/vineethk/Caisson
Sapper	programmatic	medium	no
SecVerilog	programmatic	medium	http://www.cs.cornell.edu/projects/secverilog/
HyperFlow	programmatic	medium	https://github.com/apl-cornell/ChiselFlow (ChiselFlow)
ReWire	programmatic	medium	http://mu-chaco.github.io/ReWire/

3.6 Commercial Tools

Even though they are relatively new, there is a number of commercial security verification tools for hardware design. These tools are quite similar to works in Section 3.2.

Mentor Graphics Questa Secure Check. The application is part of the Mentor Graphics Questa package. It receives RTL data and a spec for secure storage and paths. The spec is defined in Tool Command Language (TCL). 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 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 [44].

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 Jasper's 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 [19].

4 SECURITY VERIFICATION FOCUSING ON SOFTWARE LEVELS OF A SYSTEM

The second class of projects that our survey deals with focuses on verifying the security properties of software, while considering the ISA or a machine model of the hardware. Here, we investigate how the security of software is verified in the literature with a hardware model, e.g., 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 the scope of this survey. For software-only security verification, we refer the reader to the following surveys [10, 43, 83, 92, 105].

The surveyed projects fall in two categories. First, verification with respect to ISA is where the verification process involves generating assembly code that is considered correct and embodies the program with desired security properties. Typically, assembly code has one-to-one correspondence to the ISA; thus the verification process ties the software to the hardware ISA level. Second, verification with respect to a machine model is where the verification process involves a model of the target machine, such as the memory, registers, etc. The machine model is typically very simplified, but it considers key hardware features in the verification process nevertheless.

4.1 Verification with Respect to ISA

4.1.1 seL4. [57] was the first operating system microkernel that was formally verified for functional correctness. 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. It uses capability-based security model [63] for access control to enable formal reasoning about object accessibility. seL4's implementation is formally proven correct against its specification and has been proved to enforce strong security properties.

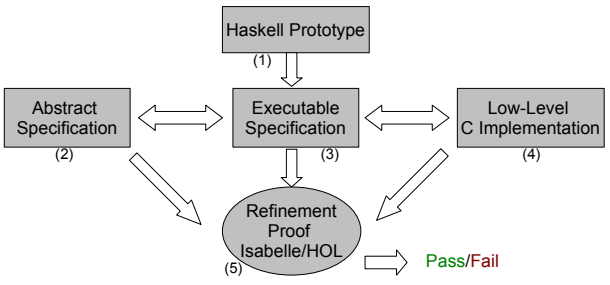

Fig. 20. The seL4 verification process.

The verification process is shown in Figure 20. In Figure 20, the system prototype is coded in Haskell according to a high-level specification (1). The specification includes a detailed functional

and behavioral description of the system (2). Isabelle/HOL theorem prover generates an Executable Specification out of the Haskell code (3). 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 implementation details and data structures that the low-level implementation must have. The last layer is the actual C implementation of seL4 (4). These three layers used in the formal verification are: abstract specification, executable specification, and C implementation, (5) in Figure 20. The total effort for SeL4 was 11 person years with 14k lines in Haskell/C and 33k lines in Isabelle. The total size of the proof is 200k including generated proofs.

4.1.2 CertiKOS and Deep Specifications. [45] 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 hides the implementation through a definition of an interface. Deep Specifications is based on the verification of abstraction layers that define interfaces to other layers hiding the implementation details. In mCertiKOS, each layer represents an abstraction, and its behavior is defined in a specification as shown in Figure 21. 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 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 [57], where the whole system is verified at once, mCertiKOS can be verified layer by layer or as a whole.

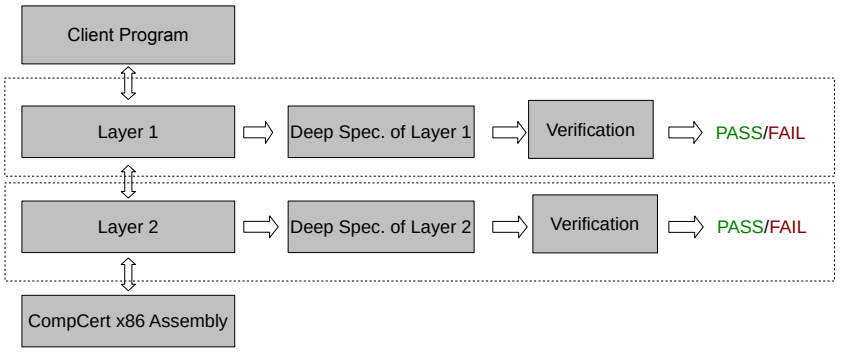

Fig. 21. CertiKOS verification process.

mCertiKOS uses two core languages for high-level and assembly-level code to describe the behavior of the system: ClightX and LAsm. ClightX is based on CompCert Clight language [15], a formally verified optimizing compiler for a large subset of the C99 programming language (known as Clight), 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 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. The whole CertiKOS took 11.5 person months to finish.

4.1.3 Verve and Ironclad Apps. Verve [107] 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 the 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) [76] and a TAL-checker [21]. The kernel is written in safe C# and the code is then compiled to TAL by a special compiler. TAL-checker is used to verify that the assembly code does not violate the primitive abstractions of the language.

Verve is still an experimental OS which lacks some modern features such as exception handling and multiprocessor support, and it assumes the hardware is trusted. 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). The specification and proof cost 5494 lines of Boogie code, while the system implementation uses 1377 instructions, resulting a 4× annotation ratio.

Ironclad Apps [50] focuses on the execution of remote applications in a secure and a functionally-verified manner. Ironclad uses Verve as the operating system. 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. However, the BIOS, and peripheral devices can be malicious. 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.

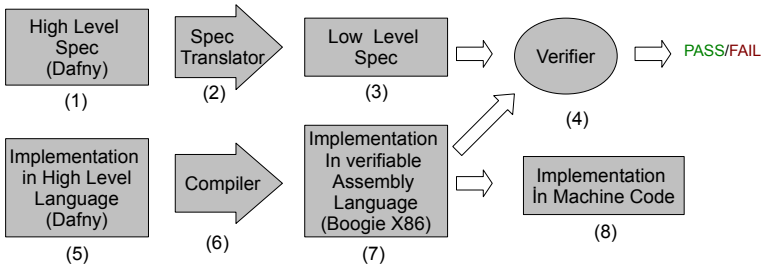

Fig. 22. Ironclad verification process.

The verification process requires an implementation in a high-level language and a high-level specification of the application code which are written in Dafny [61]. 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) [107], (5), (6), and (7) in Figure 22. Meanwhile, the high-level spec is translated into a low-level spec by a spec-translator tool, (1), (2), and (3) in Figure 22. 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 22.

4.1.4 Komodo. Even though hardware based security mechanisms provide powerful solutions, they are somehow slower to adapt new changes and to provide defenses for new security threats. For

example, Intel SGX has not been improved to provide defenses against “controlled-channel" attacks that leak information using the ability of the OS to observe page faults in enclaves. Komodo [36] provides a different approach to Intel SGX-like architectures by moving the management structure of enclaves to a privileged software monitor. Komodo is the first formally-verified, software-based implementation of an SGX-like enclave isolation mechanism [36]. Its design decouples enclave hardware primitives from security-critical but formally verified software, enabling independent evolution of the two. It employs noninterference to prove high-level guarantees of confidentiality and integrity.

The specification of Komodo including its monitor code is then formally proved that it protects the confidentiality and integrity of enclave code and data from the other software (including OS and hypervisor) running on the same machine. The proof establishes that enclave state and out-of-enclave state do not interfere with each other. As in SGX, Komodo does not prove that user code inside the enclave cannot leak information.

The implementation uses the Vale programming language [16], which consists of assembly language instructions together with annotations, such as preconditions, postconditions, and loop invariants, that describe the behavior of the instructions. The Vale generates an abstract-syntax-tree (AST) representation of the instructions and proof about the behavior of the instructions in Dafny Language [61]. Dafny uses Z3 to verify the proofs generated by Vale. A trusted assembly printer turns the instruction ASTs into GNU assembly format. A prototype of Komodo has been implemented in ARM TrustZone, since it is capable of providing its basic hardware requirements. The hardware specification covers a subset of the ARMv7 architecture.

4.2 Verification with respect to a Machine Model

4.2.1 SecVisor. SecVisor [42, 88] is a hypervisor designed to provide execution and code integrity. It guarantees that code can execute in kernel mode only if the code is approved by the user, and the 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 hardware model, a SecVisor model, and an attacker model. 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 simplify the model conservatively to avoid false negatives. So when Murφ returns with success, the SecVisor is proved to satisfy all the security properties.

To model the attacker, an actual model where the attacker behavior is modeled and an ideal model without the attacker model, akin to the models used in the verification of XOM [67]. In the actual model, the attacker can write to any memory pages with the permission bits set and can update page tables. The execution integrity is the equality between the actual model and ideal model where the attacker behavior is not modeled. 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. The whole model costs 500 lines in Murφ, and takes 343.97 s to finish the model checking for models with 4-page table entries.

4.2.2 MinVisor. MinVisor [29] 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 a 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.

4.2.3 AAMP7G. The AAMP7G microprocessor [106] provides "Intrinsic partitioning", where each partition has exclusive time slices of CPU execution, and exclusive memory space. The time and space partitioning is achieved by its "separation kernel" in microcode. To verify separation kernel, a formal security specification abstractly describing the separation kernel, and a microcode-level functional design model closely corresponding to the implementation are built in ACL2. The entire AAMP7G model is about 3000 lines of definitions. The National Security Agency evaluation team conducts a code-to-spec review to validate the microcode-level model. It is then proved that an abstract model enforces the security specification, and the microcode-level corresponds to the abstract model. The strict partition is formally verified in ACL2. Furthermore, a formal model of the instruction set is built, which can be used for analysis of user programs.

4.2.4 Verification of Noninterference at ISA Level. Fox [41], who improved upon work of Myreen, et al. [77], presented a framework for decompilation of machine (assembly) code into statements that can be processed by the HOL4 interactive theorem prover. One of the contributions of [41] is design of a domain specific language, L3, to describe the properties of an ISA. L3 can be converted to statements which can be processed by HOL4. Another contribution is the definition of numerous instruction behaviors of ISAs in L3. Later, Schwarz, et al. [87], derive noninterference properties of ARM and MIPS ISAs using the ISA definitions from [41]. Their framework determines automatically which system components (e.g., program counter or status registers) are accessible at given privilege level, based on the ISA definition. Noninterference is proved by checking how different components (e.g., status registers used by a given instruction) affect state or any return value of an instruction. For the verification, the user has to manually label certain components as "low", such as the program counter is low. Then the tools check all possible instructions from the ISA to determine which components can affect the "low" component, and these components are themselves re-labeled as "low." At the end, the tools output which components should be considered as "low", given the initial specification.

4.2.5 XMHF. XMHF [104] is an extensible and modular hypervisor. The focus of verification is to preserve the fundamental hypervisor security property of memory integrity (i.e., ensuring that the hypervisor's memory is not modified by software running at a lower privilege level). To verify the memory integrity, security invariants are inserted into the C code as assertions. However, the full functional correctness is not verified. 5208 lines of the C code is verified automatically by CBMC model checker [59], while the remaining 422 lines of C and 388 lines of assembly are manually audited.

4.3 Tools Automatically Converting Software to System Models in Verification Tools

Tools are also developed to convert the system implementation and automatically insert assertions for verification. Many architectures provide security features like isolated memory regions, e.g., ARM TrustZone, Intel SGX, and AMD memory encryption. In the following, there are two examples that verify the security of an application with the security feature provided by hardware.

4.3.1 MOAT. MOAT [94] 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) [2]. 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.

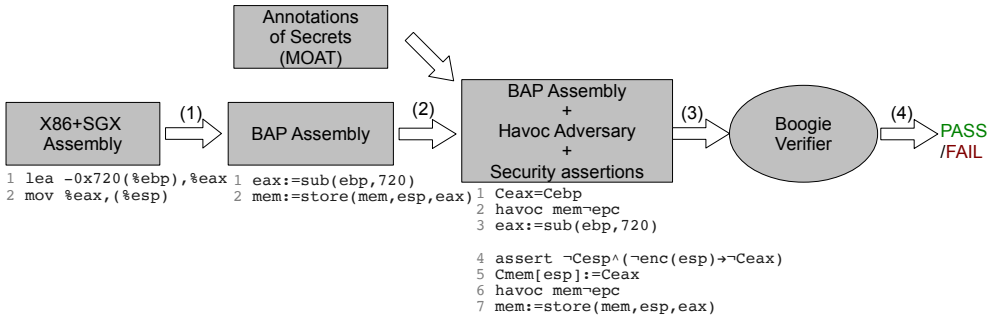

Fig. 23. MOAT verification toolchain.

The input of MOAT is the x86+SGX assembly code of an enclave user program, (1) in Figure 23, alongside with annotations indicating the location of secret data. The usage of assembly code as input to the verification process eliminates the need for a trusted compiler. MOAT then translates the assembly code to BAP (Binary Analysis Platform) assembly, which is a simple, RISC-like instruction set [17], as shown by (1) in Figure 23. 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 location 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 symbolically modifies all the non-enclave memory after every instruction of the enclave code, and is able to 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 23. 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 23, 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 23. 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 23.

MOAT provides a methodology to prove the security properties of software developed for the Intel SGX architecture. Several applications such as One-time password (OTP) service, query processing over encrypted databases are verified as an example. The query processing enclave code,

consisting 575 instructions, needs 9 policy annotations and takes 55 sec to proof. It is also the first work to create a formal model of Intel's new SGX instructions.

4.3.2 SIR. Another similar work [93] considers the applications in containers that provide isolation, referred to as Secure Isolated Regions (SIR), such as SGX. It presents a methodology for designing them to certify that code and data in SIR remain confidential. This approach decomposes an application to user code (U) that implements the functionality of the application and a small runtime library (L) that provides a narrow interface between U and the untrusted platform outside SIR. It compiles the user code with a compiler that inserts run-time checks that aid verification, and linking it with a verified runtime that implements secure communication channels. The focus of the work is to prove the confidentiality of the U running in SIR by verifying that U satisfies the "WCFI-RW" property (weak form of control-flow integrity along with restrictions on reads and writes).

Different from MOAT where annotations from programmers are needed for fine-grained information flow tracking in the application memory, this work requires U to perform communication with outside SIR only through a narrowed constrained interface provided by L, and everything in U's memory is considered confidential. This is thus a more modular and scalable approach compared to MOAT.

This work first uses a compiler to generate machine code of U with runtime checks to guarantee WCFI-RW. To model the x86 and SGX assembly code, BAP assembly [17] is used. As the compiler is not trusted, it can further optimize out or modify the runtime checks. Therefore, the final assembly generated from the compiler is taken as input to the verification. In the syntax of U code, havoc statements are used to model the action of adversary on memory variables. A static verifier generates proof obligations for each instruction in the procedure by inserting assertions. These static assertions are then discharged automatically by an SMT solver, here is the Boogie verifier.

Three large MapReduce examples are verified and evaluated in their work. The overhead of the runtime checks is 15% on average, and the static verification takes less than 20s.

5 SUMMARY AND CONCLUSION

Tables 3 and 4 present a summary of the main projects reviewed in Sections 3 and 4, respectively. 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 to help summarize the projects for the readers of this survey. We also highlight some take-away lessons from the tools and approaches reviewed in this survey.

Most of the projects surveyed use general-purpose verification tools, as shown in the *Tool* column of Tables 3 and 4. 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 is done. Security specifications are often described as formulas in theorem provers like Coq, or as invariants in model checkers like Murφ, as shown in the *Specification* column of Tables 3 and 4. Often, a model (system representation) that is separate from the system implementation (actual system) is required, e.g., Micro-Policies [30], Cache verification [113], XOM [68], SecVisor [42]. Designers have to make sure their model accurately mirrors the system implementation, otherwise the result of verification might not be correct.

Meanwhile, some of the projects take the 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 [14],

Dafny [61], or TAL-compiler [28]. 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. Alternatively, 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 3 and 4 to make sure the generated design has the desired security properties, such as Sapper [64], Caisson [65], and SecVerilog [111].

Confidentiality and integrity are the two main security properties often sought in a system. The verification of computer system often then covers these properties, but the security checks can also 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 3. Software projects, as illustrated in Table 4, have a wider variety of verification aspects, which try to verify confidentiality or integrity, but only within the selected levels. Designers generally try to provide verification of integrity or confidentiality for a system given the defined threat model. For example, SecVisor [42] verifies execution and code integrity which is a subsection of the whole memory.

The Trusted Computing Base (TCB) often encompasses multiple levels of the system from hardware to software. However, as can be seen in Tables 3 and 4, verification projects are typically focused on the hardware levels, or focused on the software levels. Bringing the hardware and software levels together is difficult but 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 [102], Intel SGX [73], and AMD memory encryption[1]. The working of this hardware with software needs to be verified for security, and requires spanning many system levels.

To conclude, 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, remains 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 the introduction of security-focused hardware, such as Intel SGX. Trusting remote software and hardware is more critical now than before, as it handles users' ever-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.

Table 3. Summary of projects that focus on hardware verification. These projects were detailed in Section 3.

Name	System Representation	Verification Tool	Custom Tool	Levels								Verification Method	Verification Aspect
				App	OS	Hypervisor	ISA	uArch	RTL	Gate	Physical		
Micro-Policies [30]	Coq Language	Coq IDE	none				✓					Theorem Prover	Non-interference, sealing, etc.
Cache Verification [113, 114]	Murφ Language	Murφ [33]	none					✓				Model Checking	Confidentiality, Integrity
XOM [68]	Murφ Language	Murφ	none					✓	✓			Model Checking	Confidentiality, Integrity
CheckMate [100, 101]	Alloy Language	Alloy [54]	none					✓				Model Checking	Covert Channels
VeriCoq [14]	Verilog	Coq IDE	VeriCoq					✓	✓			Theorem Prover	Information Flow
Formal-HDL [46]	VHDL	Coq IDE	VHDL converter					✓	✓			Theorem Prover	
RTLIFT [85]	Verilog	none	RTLIFT Tool						✓			Model Checking	Information Flow
UPEC [35]	Chisel Language	none	UPEC Tool					✓	✓			Model Checking	Covert Channels
RTL-ConTest [75]	Chisel Language	none	RTL-ConTest Tool						✓			Concolic Tesing Theorem Prover	Information Flow
Coppelia [112]	Verilog	KLEE [18]	Coppelia Tool						✓			Backward Symbolic Execution	CPU security assertions
Caisson [65]	Cassion Language	none	Cassion Tool					✓	✓			Information Flow Tracking	Non-interference
Sapper [64]	Sapper Language	none	Sapper Tool					✓	✓			Information Flow Tracking	Non-interference
SecVerilog [111]	SecVerilog Language	Z3 [31]	SecVerilog Tool					✓	✓			Information Flow Tracking	Non-interference
HyperFlow [40]	ChiselFlow Language	Z3 [31]	ChiselFlow Tool					✓	✓			Information Flow Tracking	Non-interference
ReWire [84]	ReWire	Haskell	ReWire Tool					✓	✓			Theorem Prover	Non-interference

Table 4. Summary of projects that focus on software verification with respect to the ISA or machine model. These projects were detailed in Section 4.

Name	System Representation	Verification Tool	Levels					Verification Method	Verification Aspect	Verification Effort
			App	OS	Hypervisor	ISA	μArch			
SeL4 [57]	C, Haskell	Isabelle/HOL		✓				Theorem Prover	Functional Correctness, Capability-based Security	200k lines of Isabelle to verify 8700 lines of C code in 22 person-years
CetriKOS [45]	Clight, LAsm	Coq IDE		✓	✓	✓		Theorem Prover	Functional Correctness, Non-interference	11.5 person-months
Verve [107]	TAL, C#, Boogie Lang.	Boogie/Z3 verifier, TAL checker		✓		✓		SMT solver	Type & Memory Safety	5.5k lines of Boogie in 9 person-months
Ironclad Apps [50]	Dafny	Boogie/Z3 verifier, Custom Compiler	✓			✓		SMT solver	Functional Correctness, Memory Safety	36k lines spec and proof in 3 person-years
Komodo [36]	Vale, Dafny	Boogie/Z3 verifier, Custom Translator			✓	✓		SMT solver	Non-interference	23K spec and proof in 2 person-years
XMHF [104]	C, assertions	CBMC			✓	✓		Model Checking	Memory Integrity	5208 lines of C code
SecVisor [42]	Murϕ Language	Murϕ			✓		✓	Model Checking	Execution and code Integrity	500 lines Murϕ
MinVisor [29]	ACL2 Language	ACL2			✓	✓	✓	Theorem Prover	Code Integrity	1K lines of binary code to be verified.
AAMP7G [106]	ACL2 Language	ACL2			✓	✓	✓	Theorem Prover	Non-interference	3k lines in ACL2
ISA [87]	L3	HOL4				✓	✓	Theorem Prover	Non-interference	N/A
MOAT [94]	BAP Assembly	Boogie/Z3 verifier, BAP	✓			✓		SMT solver	Confidentiality	a few policy annotations
Verification of SIR [93]	BAP Assembly	Boogie/Z3 verifier, BAP	✓			✓		SMT solver	Confidentiality	less than 20s

REFERENCES

[1] AMD. 2016. AMD Memory Encryption. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed May 2016.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[3] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1691–1696.

[4] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[5] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 165–178.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 1216–1225.

[7] Christel Baier, Joost-Pieter Katoen, et al. 2008. *Principles of model checking*. MIT press Cambridge.

[8] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[9] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[10] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2013. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media.

[11] Josh Berdine and Nikolaj Bjørner. 2014. Computing all implied equalities via SMT-based partition refinement. In *Automated Reasoning*. Springer, 168–183.

[12] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

[13] M. M. Bidmeshki and Y. Makris. 2015. Toward automatic proof generation for information flow policies in third-party hardware IP. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. 163–168.

[14] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation. In *International Symposium on Circuits and Systems (ISCAS)*. IEEE, 29–32.

[15] Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288.

[16] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*. 917–934.

[17] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. *BAP: A Binary Analysis Platform*. Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469.

[18] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[19] CADENCE. 2016. JasperGold Security Path Verification App. url-http://www.cadence.com/products/fv/jaspergold_security/pages/default.aspx.

[20] David Champagne and Ruby B Lee. 2010. Scalable architectural support for trusted software. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.

[21] Juan Chen, Chris Hawblitzel, Frances Perry, Mike Emmi, Jeremy Condit, Derrick Coetzee, and Polyvios Pratikaki. 2008. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 183–192.

[22] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 93–107.

[23] Edmund Clarke, Orna Grumberg, and D Long. 1993. Verification tools for finite-state concurrent systems. In *A decade of concurrency reflections and perspectives*. Springer, 124–175.
[24] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2011. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*. Springer, 1–30.
[25] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086 https://eprint.iacr.org/2016/086.
[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2015. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. Cryptology ePrint Archive, Paper 2015/564. https://eprint.iacr.org/2015/564 https://eprint.iacr.org/2015/564.
[27] Thomas M Cover and Joy A Thomas. 2012. *Elements of information theory*. John Wiley & Sons.
[28] Karl Crary. 2003. *Toward a foundational typed assembly language*. ACM.
[29] Mike Dahlin, Ryan Johnson, Robert Bellarmine Krug, Michael McCoyd, and William Young. 2011. Toward the verification of a simple hypervisor. *arXiv preprint arXiv:1110.4672* (2011).
[30] Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–830.
[31] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[32] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. 2014. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*.
[33] David L Dill. 1996. The Mur ϕ verification system. In *Computer Aided Verification*. Springer, 390–393.
[34] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[35] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. 2019. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 994–999.
[36] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 287–305.
[37] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. 2017. Secure information flow verification with mutable dependent types. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 1–6.
[38] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. *Lightweight Verification of Secure Hardware Isolation Through Static Information Flow Analysis*. Technical Report. Technical report, Cornell University.
[39] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 555–568.
[40] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. 2018. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1583–1600.
[41] Anthony Fox. 2015. Improved tool support for machine-code decompilation in HOL4. In *International Conference on Interactive Theorem Proving*. Springer, 187–202.
[42] Jason Franklin, Arvind Seshadri, Ning Qu, Sagar Chaki, and Anupam Datta. 2008. *Attacking, repairing, and verifying SecVisor: A retrospective on the security of a hypervisor*. Technical Report. Technical Report CMU-CyLab-08-008, Carnegie Mellon University.
[43] Iván Garcıa-Ferreira, Carlos Laorden, Igor Santos, and Pablo Garcia Bringas. 2014. A survey on static analysis and model checking. In *International Joint Conference SOCO*. 443.
[44] Mentor Graphics. 2016. Mentor Graphics Questa Secure Check. https://www.mentor.com/products/fv/questa-secure-check
[45] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 595–608.
[46] Xiaolong Guo, Raj Gautam Dutta, Prabhat Mishra, and Yier Jin. 2016. Automatic RTL-to-Formal Code Converter for IP Security Formal Verification. In *Microprocessor and SOC Test and Verification (MTV), 2016 17th International Workshop on*. IEEE, 35–38.

[47] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *Journal of the ACM (JACM)* 40, 1 (1993), 143–184.

[48] Uwe Hatnik and Sven Altmann. 2004. Using ModelSim, Matlab/Simulink and NS for simulation of distributed systems. In *International Conference on Parallel Computing in Electrical Engineering*. IEEE, 114–119.

[49] Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[50] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-end security via automated full-system verification. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 165–181.

[51] Gerard J Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279.

[52] Yao Hsiao, Dominic P Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 679–694.

[53] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[54] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76.

[55] Yier Jin and Yiorgos Makris. 2013. A proof-carrying based framework for trusted microprocessor IP. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 824–829.

[56] Matt Kaufmann and J Strother Moore. 2008. An ACL2 tutorial. In *Theorem Proving in Higher Order Logics*. Springer, 17–21.

[57] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[59] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[60] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[61] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 348–370.

[62] Xavier Leroy. 2012. The CompCert C verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt* (2012).

[63] Henry M Levy. 2014. *Capability-based computer systems*. Digital Press.

[64] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 97–112.

[65] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 109–120.

[66] Sheng Liang. 1998. Modular monadic semantics and compilation. (1998).

[67] David Lie, John Mitchell, Chandramohan A Thekkath, and Mark Horowitz. 2003. Specifying and verifying hardware for tamper-resistant software. In *Symposium on Security and Privacy*. IEEE, 166–177.

[68] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices* 35, 11 (2000), 168–177.

[69] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[70] Roger Lipsett, Carl F Schaefer, and Cary Ussery. 2012. *VHDL: Hardware description and design*. Springer Science & Business Media.

[71] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*. 605–622.

[72] Derek Lockhart and Christopher Batten. 2014. Hardware Generation Languages as a Foundation for Credible, Reproducible, and Productive Research Methodologies. In *Workshop on Reproducible Research Methodologies (REPRODUCE)*.

[73] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. 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.

[74] Kenneth L McMillan. 1993. The SMV system. In *Symbolic Model Checking*. Springer, 61–85.

[75] Xingyu Meng, Shamik Kundu, Arun K Kanuparthi, and Kanad Basu. 2021. Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[76] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

[77] Magnus O Myreen and Michael JC Gordon. 2007. Hoare logic for realistically modelled machine code. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 568–582.

[78] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[79] Rishiyur Nikhil. 2004. 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*. IEEE, 69–70.

[80] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[81] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[82] Sam Owre, John M Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *Automated Deduction—CADE-11*. Springer, 748–752.

[83] Christine Paulin-Mohring. 2011. Introduction to the Coq proof-assistant for practical software verification. In *Tools for Practical Software Verification*. Springer, 45–95.

[84] Adam Procter, William L Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015. Semantics driven hardware design, implementation, and verification with ReWire. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 13.

[85] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on* 21, 1 (2003), 5–19.

[86] Bruce Schneier. 2016. The Internet of Things Will Turn Large-Scale Hacks into Real World Disaster. https://motherboard.vice.com/read/the-internet-of-things-will-cause-the-first-ever-large-scale-internet-disaster, accessed July 25, 2016..

[87] Oliver Schwarz and Mads Dam. 2016. Automatic Derivation of Platform Noninterference Properties. In *International Conference on Software Engineering and Formal Methods*. Springer, 27–44.

[88] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 335–350.

[89] Sanjit A Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating modeling, verification, synthesis and learning. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 1–10.

[90] Ofer Shacham, Megan Wachs, Andrew Danowitz, Sameh Galal, John Brunhaver, Wajahat Qadeer, Sabarish Sankaranarayanan, Artem Vassiliev, Stephen Richardson, and Mark Horowitz. 2012. Avoiding game over: Bringing design to the next level. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 623–629.

[91] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[92] Vijay D Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27, 7 (2008), 1165–1178.

[93] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P Lopes, Sriram Rajamani, Sanjit A Seshia, and Kapil Vaswani. 2016. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 665–681.

[94] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. 1169–1184.

[95] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit Seshia. [n. d.]. A Formal Foundation for Secure Remote Execution of Enclaves. ([n. d.]).

[96] Jakub Szefer and Ruby B Lee. 2012. Architectural support for hypervisor-secure virtualization. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 437–450.

[97] Donald Thomas and Philip Moorby. 2008. *The Verilog® Hardware Description Language*. Springer Science & Business Media.

[98] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

[99] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. 109–120.

[100] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 947–960.

[101] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. *IEEE Micro* 39, 3 (2019), 84–93.

[102] ARM Trustzone. 2016. *TrustZone Information Page*. Technical Report. http://www.arm.com/products/processors/technologies/trustzone/

[103] Antti Valmari. 1996. The state explosion problem. In *Advanced Course on Petri Nets*. Springer, 429–528.

[104] Amit Vasudevan, Jonathan M Mccune, James Newsome, and Cylab Carnegie Mellon. 2012. Design and Implementation of an eXtensible and Modular Hypervisor Framework. (2012).

[105] Dolores R Wallace and Roger U Fujii. 1989. Software verification and validation: an overview. *Ieee Software* 6, 3 (1989), 10.

[106] Matthew M Wilding, David A Greve, Raymond J Richards, and David S Hardin. 2010. Formal verification of partition management for the AAMP7G microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 175–191.

[107] Jean Yang and Chris Hawblitzel. 2010. Safe to the last instruction: automated verification of a type-safe operating system. In *ACM Sigplan Notices*, Vol. 45. ACM, 99–110.

[108] Jean Yang and Chris Hawblitzel. 2011. Safe to the last instruction: automated verification of a type-safe operating system. *Commun. ACM* 54, 12 (2011), 123–131.

[109] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A High Resolution, Low Noise, L3 Cache {Side-Channel} Attack. In *23rd USENIX security symposium (USENIX security 14)*. 719–732.

[110] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2012. Language-based control and mitigation of timing channels. *ACM SIGPLAN Notices* 47, 6 (2012), 99–110.

[111] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.

[112] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-end automated exploit generation for validating the security of processor designs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 815–827.

[113] Tianwei Zhang and Ruby B Lee. 2014. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 96–105.

[114] Tianwei Zhang and Ruby B Lee. 2014. *Secure cache modeling for measuring side-channel leakage*. Technical Report. Tech. Report, http://palms. ee. princeton. edu/node.