Lecture Note

Chapter 9:
Security Verification of Processor Architectures

Chapter 10:
Principles of Secure Processor Architecture Design

CYENG 225: Microcontroller Essentials for Cyber Applications

Instructor: Dr. Shayan (Sean) Taheri
Gannon University (GU)

Chapter 9 Overview

- **Major Items**
  - It forms a stand-alone chapter that is a mini survey of approaches for security verification of processor architectures.
  - It discusses motivation for the need for formal security verification.
  - It overviews the different levels (ISA, Microarchitecture, etc.) in a system and how verification can be performed at the different levels.
  - It discusses different security verification approaches.
  - It presents a mini survey of the different hardware-software security verification projects which exist today.
  - It presents assumptions relating to the security vitrification of processor architectures.

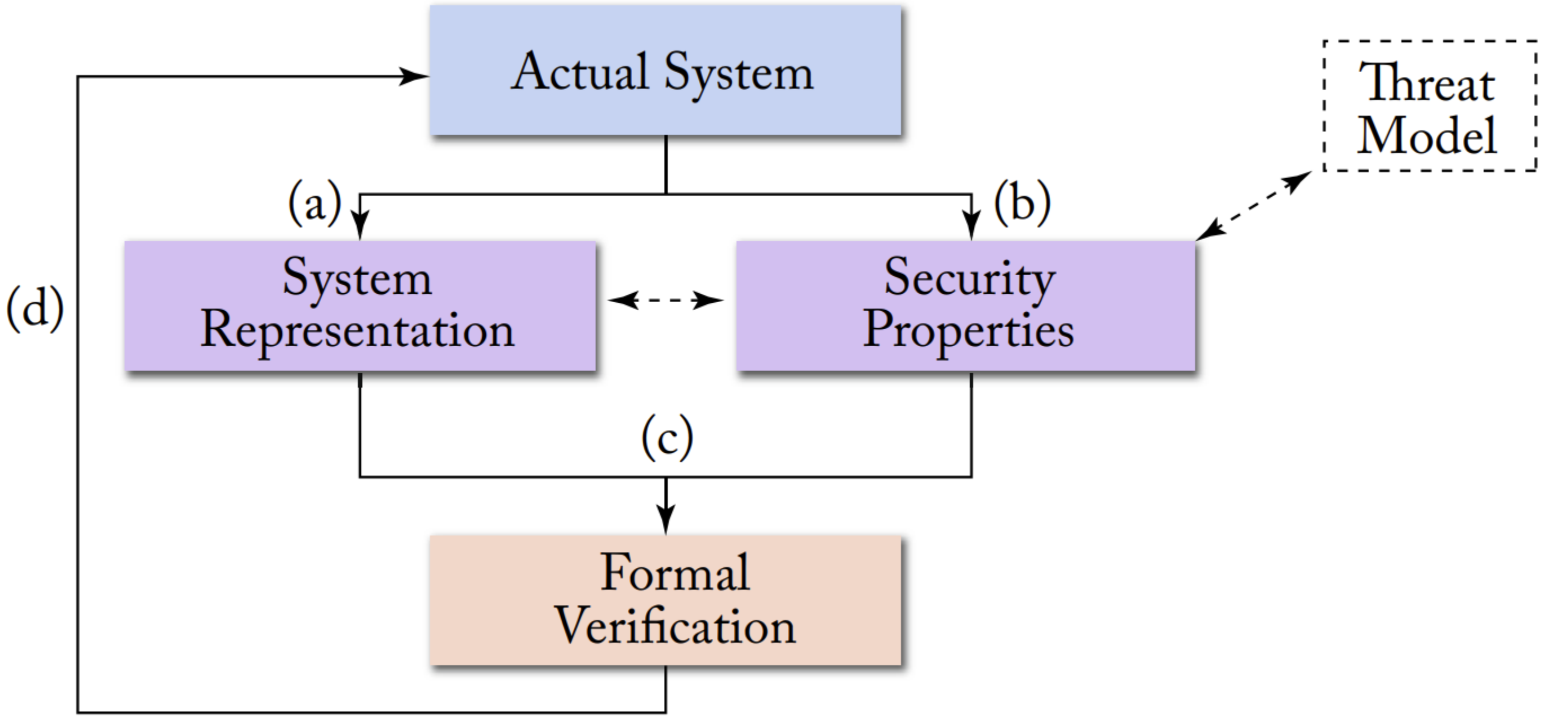
Motivation for Formal Security Verification

- 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 problems.
- If the design of the hardware, software, or the way the two interact is not perfect, then there may be security vulnerabilities that attackers can exploit.
- Many secure processor architectures have been designed in academia to provide enhanced security features in hardware.
- The proposed hardware-software system depends critically on the hardware and correct hardware-software interactions to provide the security.
- These secure architecture most often do not come with any formal specifications or proofs for security.
- In industry, increasing number of designs provide some hardware features for security, e.g., ARM TrustZone, Intel SGX, or AMD Memory Encryption.
- These designs also all rely on the assumption that the hardware is correct—the industry designs also often do not have any publicly available formal security specifications nor proofs. With formal security verification, the designers could prove the system design and implementations are secure and trustworthy.
- To help find potential security vulnerabilities and prove the designed system is trustworthy, formal methods can be used.
- Since the security of the systems depends on the correctness of the protections that both the hardware and software components provide, there is the need to verify the security of both the software and the hardware.
- Formal methods have been used extensively in the functional verification of hardware and software for a long time.
- Recently, the use of formal methods for verifying the security features of the hardware and the software levels of computing systems has emerged as an important research topic.

Security Verification Approaches

- The starting point of the verification 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 parallel, the security properties of the system need to be specified.
- The security properties are closely tied to the system’s assumed threat model. Security verification requires agreement on a threat model.
- If the threat model agreed upon does not represent the reality or the user needs, then the verification results may not be meaningful.
- Assuming the threat model is agreed upon, the security properties can be specified separately or together within the representation of the system.
- 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.
- If the verification fails, the design needs to be updated and re-evaluated.

Security Verification Approaches (Cont.)



General procedure for security verification.

System Representation

- In order to check if a system complies with some properties, a representation of the system that accurately expresses its behavior is needed.
- Ideally, the actual system description can be used, such as the hardware description language (HDL) source code for hardware components, or a programming language source code for software components.
- Otherwise, a model of the hardware behavior in the verification tool is needed.
- One reason a model may be needed is that the way a system is described in HDL or programming language may not be compatible with the verification tool that is being used, or the way the system is described is too complex for the verification process to handle.
- There is a danger, however, that the model may not accurately represent the actual system.
- If an automated method of creating a model (from source code representation) is not available, then model has to be created manually by engineers.
- When creating models manually, proving the correspondence between the model and the actual system is an open research problem.
- Hardware components can be described with Hardware Description Languages (HDLs).
- The most popular HDLs are Verilog and VHDL.

System Representation (Cont.)

- More high-level abstractions and reusability, than with traditional HDLs, is offered by hardware generation languages (HGLs) or hardware construction languages (HCLs), such as Chisel, Bluespec, or Genesis2.
- These languages usually focus on functional specification, but can be extended to include security related information, usually via annotations in the code.
- Software components can be described by their high-level implementation in programming languages such as C, CCC, or Java.
- Like with hardware languages, system specification can be integrated with security-property specification, and include security verification related information inside the code itself as annotations.
- Given the system representation, formal verification is done with respect that system representation.
- Most projects assume a trusted compiler or tool chain such that the system realization indeed matches the system representation, and does not contain extra hidden, or unwanted, functionality that may compromise the security of the system.
- A malevolent compiler might insert malicious code into the binary, where a virus-infected compiler as able to inject backdoors into applications during compilation.
- Researchers have developed trusted compilers that are guaranteed not to inject behavior that was not specified.
- A system representation is needed to verify the system.
- Actual source code, or a model of the system, can serve as the representation.
- Even a verified system, however, is susceptible to vulnerabilities if the real system does not match the source code or the model - this may be due to attacks at manufacturing time or untrusted toolchains.

System Properties

- The main security properties that designers may be interested in are the confidentiality, integrity, or availability.
- Recall, confidentiality concerns prevention of the disclosure of sensitive information to unauthorized entities.
- If a system has some registers that should not be modified except by the hardware, then these registers require confidentiality protection form the system levels above the hardware.
- Integrity concerns prevention of unauthorized modification of sensitive information.
- If certain data should not be modified except by the operating system, then it requires integrity protection from the system levels above the operating system.
- Availability concerns ensuring provision of services to legitimate users when requested.
- Availability almost never can be achieved through use of one single secure processor design.
- Availability, however, can be achieved by using many secure processor together, through redundancy.
- A typical system is broken down into many hardware and software layers.
- Often there is a linear relationship in that hardware or software components are protected from all levels above a certain level, which is least privileged level to be able to access (confidentiality) or modify (integrity) of that hardware or software component.
- Confidentiality, integrity and availability of each system component that is part of the trusted computing base needs to be considered when specifying security properties for verification.
- Any unverified component could become a source of vulnerability.

Formal Verification

- Typical formal verification mechanism include theorem provers or model checking.
- When using theorem proving approach, the security properties can be represented in terms of logical formulae.
- A logical formula serves as a limitation on the states the system is allowed throughout its execution.
- When using model checking, security properties can be expressed as invariants within a system, and their validity is checked against all possible execution paths of the system.
- Theorem provers, also called proof assistants, use formal proofs to verify that a system complies with some given properties.
- Theorem provers aid the verification process by providing frameworks for creating a mathematical model of the system, for specifying the security properties, and for formally proving whether the model complies with the properties or not.
- Theorem provers are generally composed of a language, and an environment for describing the proofs (such as CoqIDE).
- There is a number of proof assistants used actively in academia and industry such as: Coq, Isabelle/HOL, PVS (Prototype Verification System), ACL2 (A Computational Logic for Applicative Common Lisp), and Twelf (LF).
- 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 this method.
- Model checkers, on the other hand, typically use an algorithmic search, which is performed over a system's representation and its states, rather than using deduction.
- Internally, model checker often rely on satisfiability modulo theories (SMT) solvers to analyze the state space.
- In general, checking whether a system complies with a given specification is an undecidable problem.

Formal Verification (Cont.)

- With model checkers, the problem is converted into a search problem with a reasonable coverage of input instances, however, it may not always give a solution.
- Especially, model checking has a well-known state explosion problem, which is the exponential growth of the states of the system that should be evaluated.
- State explosion leads to extremely long run times or exhaustion of the compute resources, thus preventing it from giving a solution sometimes.
- For fairly complex systems, model checking needs to use more abstraction and simplified models.
- However, as the level of abstraction gets higher, there is the risk of missing some important details of the system being verified.
- The security property that is being verified using model checking has to be defined using a logical form.
- The checks can be done either for each transition or each state using invariants, pre- and post-conditions.
- The output can be positive (property satisfied), negative (property violated), or the execution runs indefinitely.
- Finite state reachability graphs are the most common way of modeling systems.
- Theorem provers or model checkers can be used for security verification of the secure processor architectures.
- Theorem provers have a much higher learning curve, meanwhile model checkers suffer from state explosion problem and in some cases may not be able to give a definite answer about security of a system.

Chapter 10 Overview

- It presents the principles of secure processor architecture design which have been derived based on the observations of best practices for secure processor design.
- It gives a short description of how the secure design principles affect the standard computer architecture principles.
- It overviews threats to the assumptions that have been discussed.
- It presents common pitfalls and fallacies of secure processor architecture design.
- It provides a list of challenges that secure processor architects can encounter, a list of future trends, and a brief note on the art and science of secure processor architecture design.

The Principles

- For secure processor architecture, there are a set of five principles that can be followed to achieve a secure design:
    1. Protect off-chip communication and memory.
    2. Isolate processor state between TEE execution.
    3. Measure and continuously monitor TCB and TEE.
    4. Allow TCB introspection.
    5. Minimize the TCB.

Protect Off-chip Communication and Memory

- Due to danger of attacks on external components, the secure processor chip assumption should be followed and any and all data and communication leaving the processor chip needs to be protected.
- It should be protected for confidentiality, integrity, access pattern, authentication and freshness.
- Encryption ensures protection of any secret or sensitive information (data or code) and should be used whenever data leaves the chip.
- Encryption should be used by the hardware and software parts of the TCB. Any memory location that is off-chip should be encrypted and hashed (using hash trees).
- Secure hash can be used to compute a fingerprint of piece of code and data stored outside the chip; the fingerprint can be used to ensure integrity.
- Randomized or oblivious accesses prevent attacks that use access patterns to learn information.
- All the off-chip communication and data needs to be authenticated with MACs or digital signatures, and freshness needs to be ensured (with proper use of nonces) so that attackers cannot use replay attacks against the system.

Isolate Processor State Between TEE Execution

- Within the processor, the hardware and software TCB create the TEE wherein protected software executes.
- Each TEE software instance shares the processor hardware with other instances of TEE software, and the untrusted software as well.
- Due to the shared hardware, numerous side channels or information leaks are possible.
- Any processor state should be isolated between execution of different TEE and also the untrusted software.
- The state includes caches, but also any buffers and memory structures in the processor, even if they are not visible to the programmers (such as the state of the branch predictor, for example).
- The state can be isolated through state flushing, where each time execution switches, all the state is securely stored, and flushed from the processor’s functional units.
- State in other components, such as I/O devices, needs to be flushed as well.

Measure and Continuously Monitor TCB and TEE

- Measurement allows to attest the system and prove it is working correctly.
- Hashes and active measurements form basis of attestation.
- However, a reference hash or measurement value is always needed to be able to make a judgement about whether the state or behavior of the system is correct or not.
- Such reference values need to be provided for all TCB and TEE components.
- Attestation also may require use of public key infrastructure (PKI), thus key management and structure of the PKI needs to be designed when the secure processor architecture is designed.
- Static hashing of code and data needs to be augmented with active measurement of the operation of the system to detect any run-time changes.
- Measurement can capture behavior of the system and detect any deviations.
- This should be applied both to the TCB itself, and the protected code and data in the TEE.

Allow TCB Introspection

- Transparency, or avoiding so-called security through obscurity, is needed to allow users to know what the system does.
- Also, attackers are often very clever, and a designer should not depend on hidden functionality to protect the system.
- Moreover, obscuring the functionality (e.g., making only binary code available) makes it difficult for legitimate users to be able to evaluate a system, or potentially fix a system if it is broken.
- Short-term benefits such as easier deployment is often offset if bugs are later found, exposing many systems to attacks.

Minimize the TCB

- Having minimal TCB ensures that it can be validated and made trustworthy.
- There is a correlation between the complexity of the system (hardware and software lines of code) and number of potential bugs.
- Thus, minimizing TCB is crucial to reduce bugs, and consequently potential vulnerabilities that can lead to attacks.
- Also, for verification of the system, the simpler the system, the more likely it can be formally verified.
- Any secure processor architecture relies on the TCB (i.e., a set of trusted hardware and software components) to realize the protections.
- However, a trusted component need not be trustworthy.
- A bug, or malicious modification, to a component in the TCB can render the system useless.
- Thus, formal security verification, and not just functional verification is needed.
- To enable security verification, TCB needs to remain small.

Assignment

- **Reading Assignment:**
  - Zferer, J., 2018. **Principles of secure processor architecture design**, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 9048, pp.1-175.
    - ✓ "Chapter 9: Security Verification of Processor Architectures", Pages 103-111.
    - ✓ "Chapter 10: Principles of Secure Processor Architecture Design", Pages 113-124.

Questions?