Towards Property Driven Hardware Security

Wei Hu, Alric Althoff, Armaiti Ardeshiricham, and Ryan Kastner
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093
Email: {weh040, aalthoff, aardeshi, kastner}@ucsd.edu

Abstract—**Secure hardware design is a challenging task due to the fact that security properties are difficult or impossible to model and subsequently verify using traditional hardware design tools. The "state of the art" for hardware design security relies heavily on functional verification, manual inspection, and code review to identify security vulnerabilities. This labor intensive process significantly reduces productivity while proving no guarantee that a security flaw will be identified. In this paper, we describe a property driven approach to hardware security, which allows automatic synthesis and verification of both qualitative and quantitative security properties. We address hardware security by enforcing information flow and statistical security properties. By incorporating a new security property specification language, such security properties can be specified, translated and verified using hardware design tools. We present design examples to demonstrate our property driven hardware security solution for proving isolation, detecting timing channel, eliminating hardware Trojan, and enforcing security related statistical properties.**

I. Introduction

In the past, computer security largely assumed that the underlying hardware was secure and dependable. This is clearly no longer true as there are many demonstrated attacks that take advantage of hardware specific security vulnerabilities. For example, smart cards running cryptographic algorithms leak the secret key through their power consumption [1]; web browsers disclose login credentials due to runtime variations of arithmetic operations [2]; logically isolated virtual machines running in the cloud leak private information through interactions caused by shared hardware resources [3]; a flaw in Qualcomm's TrustZone implementation was used to break Android's full disk encryption [4]; compromised hardware was used as a foothold to hack into a commercial jet [5]; and there is speculation that hardware backdoors are present in military weapons allowing them to be remotely disabled [6]. With an increasing number of hardware security flaws reported, it is clear that we have reached a time where hardware has become an attractive attack surface that can be exploited with potentially large consequences.

The "state of the art" for hardware design security relies heavily on manual inspection and code review. For example, chip manufacturers have large and growing security audit teams, e.g., Intel's Security Center of Excellence (SeCoE) and the Qualcomm Product Security Initiative (QPSI) group, that provide code reviews to hardware designs to identifying security flaws. This process is labor intensive and also significantly reduces productivity. Even worse, it provides no guarantee that a security flaw could be identified. This is largely due to the lack of effective tools supporting secure hardware design. Simply stated, it is difficult to evaluate security alongside traditional design parameters like resource usage, performance, and power for design space exploration.

Automated security evaluation of hardware designs requires more and better hardware security verification tools. These must allow for the specification of security properties and the ability to verify that the design adheres to those properties. We envision three fundamental elements for a hardware security design flow. First, we need a systematic approach for specifying security properties. Then, we need models to describe the security related behavior of a hardware design. Finally, we need techniques that use those models to verify these security properties on the hardware design.

However, these fundamental elements largely remain open research problems. We lack hardware security models to describe security related properties as is done with functional verification. Property specification languages provide a solution for specifying and enforcing functional properties (e.g., allowed values and event sequences). Unfortunately, they cannot easily be expanded to capture security related properties. There are common and widespread specifications for security properties, e.g., lattices for information flow [7] and mutual information for statistical information leakage [8]. We must integrate these into the hardware design flow in a manner that allows for their efficient verification.

In this work, we propose a property driven approach to hardware security. We discuss various types of security properties that should be enforced for secure hardware design and describe the features of a security property specification language. We demonstrate our property driven security solution using concrete design examples. More specifically, the contributions of this work are:

- Describing hardware security models based upon information flow and statistics;
- Using those information flow and statistical models to specify hardware security properties;
- Demonstrating our property driven hardware security methodology on several design examples.

The reminder of the paper is organized as follows. Section II introduces our property driven hardware security approach. In Section III, we describe different types of information flow and statistical security models and the properties that they enable. We discuss the key components of a security specification language in Section IV. Section V presents design examples to demonstrate our property driven hardware security methodology. Section VI discusses future research directions, and we conclude in Section VII.

II. Property Driven Hardware Security

The existing hardware design process can be roughly broken down into two parts: synthesis and functional verification. The synthesis process translates the hardware designs into different levels of abstraction – from an untimed functional model to register transfer level model to gate level model and finally a physical model. Functional verification insures that these synthesis processes are correct, and it determines that the hardware design meets certain functional properties.

However, functional correctness does not necessarily imply a secure system. A functionally correct design could have security flaws when the specification itself is insecure. This is the fundamental problem that we try to address, and we need more and better hardware security design tools in order to insure that the specification is secure in addition to insuring that the synthesis process is functionally correct.

Thus, we argue that *security verification* must be added as a third pillar to the hardware design process alongside synthesis and functional verification. In particular, we see a significant gap in the ability to specify security properties a la Property Specification Language (PSL) and SystemVerilog Assertions (SVA). To close this gap, we propose a property driven secure hardware design flow that incorporates security oriented languages and models that enable the specification and verification of security properties on a hardware design. Figure 1 provides an overview of our proposed hardware security design methodology. It integrates property driven security verification as an additional path alongside synthesis and functional verification.

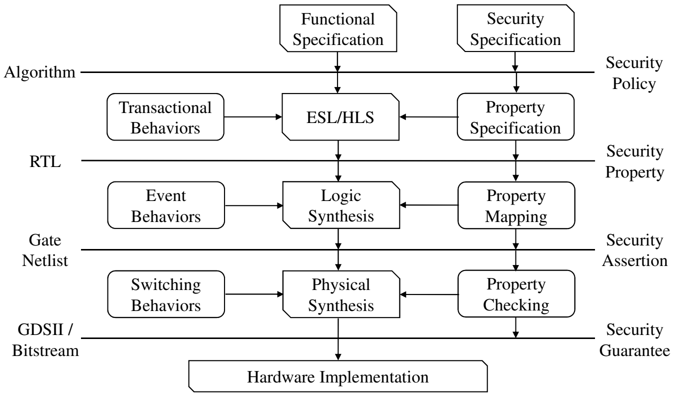

Fig. 1. We argue that the standard synthesis and functional verification process (center and left columns, respectively) must be augmented with a property driven hardware synthesis process (shown in the third/right column).

Property driven hardware synthesis starts with high level specification of the security policies that the system on chip (SoC) architecture should adhere to. These properties may be hardware-agnostic as the complete details of the SoC architecture are not fully specified. One example of a security policy could be that *CPU1 cannot read from or write to the Firmware* (see Fig. 2)

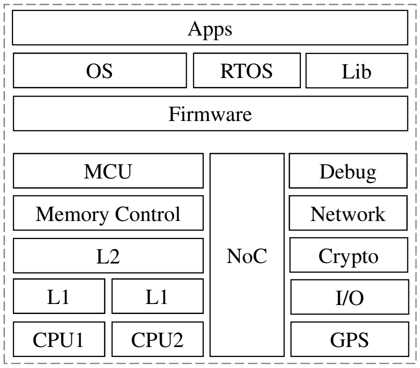

Fig. 2. A representative SoC architecture. We use this architecture to motivate the need for different hardware security properties throughout the paper.

As the SoC architecture is refined, we can similarly refine the security policy into a security property. For example, the mentioned security policy will be modified to *there should be no information flow between the memory spaces of CPU1 and Firmware*, or more formally:

```
assert iflow CPU1_MEM =/=> FW_MEM
assert iflow FW_MEM =/=> CPU1_MEM
```

When the hardware design is compiled to RTL or gate netlist, we can further map the security property to two security assertions as follows. These assertions can be instantiated along with the information flow model (described in more detail in the Section III) for the hardware design to verify for security guarantees.

```
set CPU1_MEM_taint[M:0] := 1
set default_taint := 0
assert FW_MEM_taint[N:0] == 0

set FW_MEM_taint[N:0] := 1
set default_taint := 0
assert CPU1_MEM_taint[M:0] == 0
```

There are many different security properties that one would want to check, e.g., those related to isolation/separation, integrity, confidentiality, and mitigating or eliminating side channels. We need languages to specify these security properties in a manner that is amenable to verification. At the center of this lies models that accurately encompass both the hardware design and security properties. These models ideally allow expressiveness of the properties and the ability to quickly perform formal, simulation, and emulation techniques. We identify information flow and statistical measures as a basis for two fundamental security models that are comprehensive, expressive, and verifiable.

III. Hardware Security Models and Properties

A. Information Flow Models and Properties

Information flow models rely on the notion of non-interference, which was first described by Goguen and

Meseguer [9]. It models flow relationships using a security lattice [7], and security properties can be written based upon flow relationships. One can view this as affectability or observability. For example, we can state that untrusted data should not be able to affect (flow) to the program counter. Or that confidential information should not be observable (flow) to an unprotected memory space.

Information flow analysis has a long history for modeling computer system security properties. Denning was amongst the first to take an information theoretic approach to reason about security [7]. McLean [10] and Gray [11] made significant contributions in formalizing security properties using an information flow model. Since that time, information flow analysis has been used across different layers of the computer system stack [12], [13], [14], [15] to analyze the security of the system or to ensure it behaves in a secure manner. Recent work on hardware information flow tacking (IFT) [16], [17], [18], [19], [20], [21], [22] allows expressing flow related security properties on a hardware design. There are a wide range of security properties that can be modeled using information flow. These include confidentiality, integrity, isolation/separation, timing channels, and hardware Trojans.

Confidentiality properties ensure that certain data will never flow to a publicly viewable area of the hardware. To give a few examples, the key or intermediate results in cryptographic cores should not directly flow to a point observable by an attacker (e.g., the ciphertext). Sensitive information should not flow to an untrusted IP component through interactions with shared resources (such as buses and memory). Private data in your GPS unit should not be allowed to flow to a public output, allowing an attacker to learn your location. And you would never allow your financial or biometric records to flow to somewhere that can be viewed by an unauthorized party.

Integrity is the dual property of confidentiality. Here, we mark untrusted areas of the hardware and verify that they do not affect any critical portion of the design. For example, we want to make sure that the secret key is never overwritten with unprotected data from an open communication port. We require that a untrusted IP component never modifies the firmware or that a low privileged hardware module does not interrupt critical operations.

Isolation and Separation can be enforced as an information flow security property stating that there should never be information exchange between two given hardware components. For example, the memory space for secure and non-secure IP cores should not overlap; untrusted IP cores should be isolated from the shared bus when a trusted one is transmitting information. E.g., previous work has employed gate level information flow tracking to prove isolation between trusted and untrusted IP cores in SoC designs [18].

A ***timing channel*** can also be modeled using information flow. By definition, a timing channel between two variable exists if the value of one affects the timing behavior of the other without necessarily altering its value. As an example, there can a timing channel from the private key to the ciphertext ready signal in an RSA core. The ready signal will be asserted whenever the encryption is done, but there is a timing channel if the encryption time is a function of the key or the plaintext.

Certain types of ***hardware Trojans*** can be detected using information flow security verification as well. Information flow analysis captures Trojans that leak sensitive information (e.g., the secret key) or manipulate critical data (e.g., the key register). A recent work has developed a classification of the types of Trojans that are detectable by verifying information flow security properties [22].

While information flow models are powerful for verifying a variety of security properties, they are not without drawbacks. For instance, existing information flow analysis techniques tend to provide a binary answer to security. Specifically, they indicate that no flow occurs or that there is at least one, but do not provide any indication on its severity. It is often desirable to understand a broader range of security properties, such as: is the secret key evenly mixed after a cipher block? Is the random number generator producing truly random sequences? And what is the maximum amount of information that is leaked through a side channel? Existing information flow analysis techniques are inadequate for modeling and verifying such quantitative and statistical security properties.

B. Statistical Models and Properties

Statistical measures provide a basis for models that allow describing a different type of security properties than the ones that can be done with information flow models. We motivate the employment of statistical measures as a model for security by presenting some examples of statistical security properties.

Entropy (H) describes the expected information, or uncertainty, inherent in the probability distribution of a random variable. In addition, entropy is a measure of channel capacity, in that it provides an upper bound on the amount of information that can flow through that channel. E.g., we can use it to write a property such as $\mathrm{H}(V) = X$, which states that we can learn at most X bits of information from some variable V.

Mutual information (MI) is useful for describing the amount of information that passes through a channel. MI provides a measure of the dependence between two variables, quantifying the amount of information that one variable reveals about the other. Previous work has shown that MI between the key and the encryption time has a strong correlation with the success determining the key through timing channel attacks [23]. Thus, one could use this to write a property such as the MI between the key and the encryption time should be less than Y bits, i.e., $\mathrm{MI}(key; enc\_time) < Y$.

Quantiles characterize the distributions of random variables. They are closely related to the concepts of cumulative frequency distributions and histograms. A sequence of quantiles is as useful as a histogram for providing summary statistics about a random variable except that it makes it simpler to answer questions posed in terms of order statistics, which are more robust indicators of common distribution-related inquiries. Since distribution functions are fundamental to statistical measures, quantiles enable us to build up other more complex and robust statistical models. For example, consider the security policy that a pseudorandom number generator (PRNG) behave correctly. A misbehaving random number generator can significantly compromise the security of a supposedly secure platform [24]. Thus, one would want to specify the property that a random number generator's outputs must be random, or more specifically, that its output should have some known probability distribution. A possible property to enforce this might be $\text{Quantile}(\text{PRNG}, [\alpha_0, \alpha_1, \ldots, \alpha_n]) = \text{Uniform}(0, 1, \text{'Quantile'}, [\alpha_0, \alpha_1, \ldots, \alpha_n])$, specifying that n quantiles, $\alpha_0, \alpha_1, \ldots, \alpha_n$, of the observed distribution must match those of a uniform distribution.

IV. Security Property Specification Language

In order to prove that a hardware design is secure, one must formally describe the properties it should adhere to. These properties must be specified in a formal and succinct manner. And this must be balanced with expressibility and usability of the property specification language. Specifically, the language must be usable by both security experts and hardware designers. The properties can be easily expressed by anyone, not merely security experts. And hardware designers should be able to understand and refine the properties.

Ideally, we have something similar to an existing hardware specification and assertion language. The security experts and hardware designers will use the language to state security policies, properties, and assertions. The language needs a method to specify sensitivity levels, affectability, and observability. It should also support quantifiers for measuring statistical properties. And finally, it should be easily amenable to use existing verification techniques to show that the design adheres to the specified security policies and properties. The following provides an example language that specifies security properties related to the SoC design in Fig. 3.

```
set sensitivity Firmware, Crypto, GPS := TRUSTED
set sensitivity RAM, Ethernet, JTAG := UNTRUSTED
set sensitivity CPU2, Firmware, Crypto := PRIVILEGED
set sensitivity MCU := UNPRIVILEGED
set quantifier Crypto_time := TIMER

assert isolate(TRUSTED, UNTRUSTED)
assert isolate(PRIVILEDGED, UNPRIVILEGED)
assert isolate(JTAG, Firmware) when !Debug_mode
assert noflow(Crypto_key, Ethernet)
assert H(Crypto_time) <= 0.5
assert MI(Crypto_key, Crypto_time) <= 0.2
```

We write properties that classify the cores into different security classes (E.g., TRUSTED, UNTRUSTED, PRIVILEGED and UNPRIVILEGED). We then use the language to specify allowed information flows between different cores. We insert quantifiers (e.g., a timer to measure encryption time) to enable statistical properties related to timing channels. Finally, we use the language to specify the security properties to be checked, e.g., the TRUSTED (PRIVILEDGED) IP cores should be isolated from the UNTRUSTED (UNPRIVILEGED) ones; JTAG cannot access the Firmware out of debug mode; the Crypto_key should not flow to the Ethernet; the entropy of encryption time measurements should be less than 0.5 bits; and the mutual information between the secret key and encryption time measurements should be less than 0.2 bits etc.

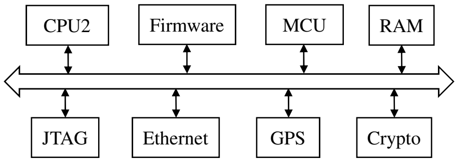

Fig. 3. A refinement of part of the SoC design from Fig. 2 which is used to demonstrate example security properties.

V. Design Examples

A. Isolation and Separation

In an SoC design, the IP cores may be designed by IP vendors of different trust. As an example, IP cores designed by a third-party are usually considered untrusted while those developed in-house are regarded as trusted. Furthermore, it is usually desirable to put the trusted IP components in a secure domain and isolate them from untrusted components. The concept is similar to what is enforced by the *ARM TrustZone* [25]. Figure 4 shows such a design example.

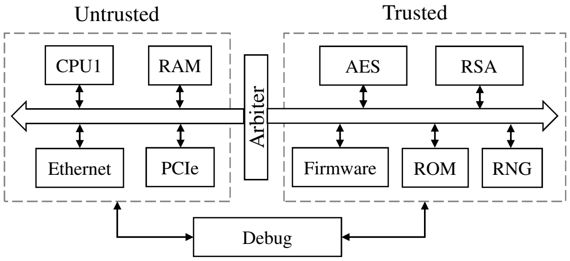

Fig. 4. Isolating trusted IP components from untrusted.

In this example, we separate the *AES*, *RSA*, *Firmware*, *ROM* and *RNG* cores into a trusted computing environment and isolate them from those connected to an open network. Here, we need to tightly control the information flows between the untrusted and trusted environments. For confidentiality, we should not allow sensitive data from the *Firmware* or *ROM* to leak to any untrusted IP component; the *CPU1* is only allowed to read from the output data port of the *AES* and *RSA* cores. For integrity, the untrusted cores are not allowed to write to the *Firmware*; the *PCIe* core is not allowed to access the address space assigned to the *AES* and *RSA* cores; *CPU1* can only write to specific addresses of the *AES* and *RSA* cores; and the *Debug* port is only allowed to update the *Firmware* when in debug mode. These can all be enforced by specifying and verifying properties related to information flow.

B. Timing Channel

The RSA cipher algorithm contains a conditional branch structure that can cause variations in encryption time, i.e., the time when the ciphertext *ready* signal is asserted. Figure 5 shows an RSA implementation. Previous work has demonstrated that there can be a timing channel in this implementation, which allows the attacker to recover the private key through statistical analysis of encryption time [26]. Such timing channel can be captured by proving information flow and statistical security properties.

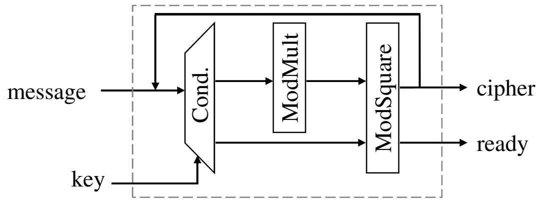

Fig. 5. A serial RSA implementation that has a timing channel.

One security property that we need to check is that there is no information flow from the *key* to the *ready* signal. This guarantees strong non-interference between the key and the ready signal. If the property fails to hold, then there can be a timing channel, and we may wish to understand the severity of this leakage. In this case, we could specify a statistical security property stating that the entropy of the encryption time under different keys and plaintexts is minimal, which will depend upon the risk tolerance of the system.

C. Hardware Trojan

Hardware Trojans are malicious design modifications that are designed to trigger under rare conditions, and thus be difficult to detect using functional verification. Figure 6 shows an AES core from *Trust-HUB* [27]. The core contains a hardware Trojan that modulates the key to leak through the unused pin *Antena*[1]. Functional testing will likely fail to detect this Trojan since it is activated after $2^{129} - 1$ successive encryption operations.

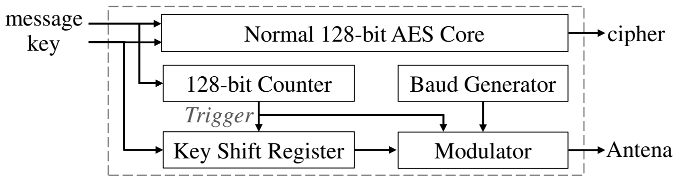

Fig. 6. A hardware Trojan design from *Trust-HUB*.

However, the Trojan design can be captured by proving an information flow security property stating that the key should never flow to *Antena*. Generally, information flow models are effective in detecting Trojans that cause violation of confidentiality or integrity properties [22].

D. Random Number Generation

Figure 7 shows a Johnson noise based random number generator used by Intel [28]. We may wish to check if the random number sequence is uniformly distributed, and thus behaving properly. This can be measured using a statistical property, e.g., the entropy of the random sequence. In addition, we may want to make sure that the random sequence will not become biased under a disturbance (e.g., thermal variation). A possible solution is to add a real-time monitor to measure the probability distribution function of the outputted random numbers. This could be done using a set of small and efficient quantile estimators.

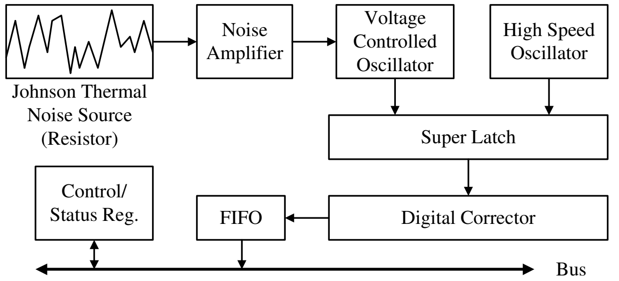

Fig. 7. A Johnson noise based random number generator by Intel [28].

VI. Potential Research Directions

We identify future research directions in developing models, languages, and tools that allow for the efficient specification and verification of security properties on a hardware design.

A. Improve Information Flow Models

A first research direction is to improve the quality of existing information flow models and establish new models that support a wider range of security properties.

Precision and Complexity Tradeoffs: Precision and complexity of information flow models are two dominant (but contradictory) factors on security verification performance. Developing effective methods for complexity and precision tradeoffs can see benefits in accelerating security verification.

High-level Information Flow Models: High-level models are usually more expressive and efficient for testing and verification. Constructing information flow models for SystemVerilog and high-level synthesis will allow more general security properties be verified during an earlier design phase.

Quantitative Information Flow Models: Quantitative models allow finer assess of security. Creating models for pertaining the severity of the flow, e.g., precisely measuring the amount of information flow and proving boundaries on information flow, is an interesting research topic.

B. Derive Statistical Security Models

We need effective statistical measurements for security and efficient methods for calculating these measures.

Statistical Security Measurements: We need more statistical measurements for different security properties, e.g., probability distribution functions, statistical distances and correlation measures. This is an important step for deriving generic parameters for measuring hardware security.

Statistical Security Measure Estimation: Designing estimators for precise estimation of statistical security measures

[1] Note that this is the way the signal is declared in the benchmark.

with a minimum number of samples is an important research direction. It also can beneficial to develop light-weight online estimators that can detect security violations in real-time.

C. Develop Security Property Specification Languages

The following research vectors are essential for developing hardware security specification languages:

Security Attributes: We need to formalize a minimum set of security attributes from various types of security properties. Such a set should be semantically complete for describing sensitivity, privilege, affectability, observability, statistical distribution and quantitative information flow. This will help define the basic elements (e.g., key words, operators and semantics) of the security property specification language.

Security Properties: The security specification language should support a wider range of properties. Apart from information flow and statistical properties, it should be able to model modes of operation (e.g., debugging, secure boot, and normal mode), complex global properties (e.g., access policies rules in SoCs and NoCs) and firmware properties (e.g., different configurations).

Security Property Compilation: There is a gap between the semantics of security specification languages and hardware security models. We need to define rules and develop tools that allow security properties be automatically translated and mapped to security models in order to be verified.

VII. Conclusion

We provide a property driven approach to hardware security. The proposed approach enforces hardware security by specifying and checking information flow security properties related to confidentiality, integrity, isolation/separation, side channels, and hardware Trojans as well as statistical security properties such as entropy and mutual information. The ultimate goal is to develop security property specification languages and hardware verification tools that allow efficient formal analysis, simulation, and emulation of these security properties for secure hardware design.

Acknowledgment

This work was supported in part by the National Science Foundation under grants CNS-1527631 and CNS-1563767.

References

[1] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Examining smart-card security under the threat of power analysis attacks," *IEEE Trans. Comput.*, vol. 51, no. 5, pp. 541–552, May 2002.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *IEEE Symposium on Security and Privacy*, May 2015, pp. 623–639.

[3] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. ACM, 2011, pp. 29–40.

[4] "Extracting qualcomm's keymaster keys - breaking android full disk encryption," in *http://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html*, June 2016.

[5] R. Waugh, "Could a vulnerable computer chip allow hackers to down a boeing 787? 'back door' could allow cyber-criminals a way in," May 2012, http://www.dailymail.co.uk/sciencetech/article-2152284.

[6] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, no. 5, pp. 34–39, May 2008.

[7] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[8] J. K. Millen, "Covert channel capacity," in *1987 IEEE Symposium on Security and Privacy*, April 1987, pp. 60–66.

[9] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20.

[10] J. McLean, "Security models and information flow," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*. IEEE, 1990, pp. 180–187.

[11] J. W. Gray III, "Toward a mathematical foundation for information flow security," *Journal of Computer Security*, vol. 1, no. 3, pp. 255–294, 1992.

[12] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan 2003.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGOPS Oper. Syst. Rev.*, vol. 38, no. 5, pp. 85–96, Oct. 2004.

[14] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 321–334, 2007.

[15] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009, pp. 109–120.

[16] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Design Automation Conference (DAC)*, June 2011, pp. 254 –259.

[17] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *the 38th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2011, pp. 189–200.

[18] R. Kastner, J. Oberg, W. Hu, and A. Irturk, "Enforcing information flow guarantees in reconfigurable systems with mix-trusted IP," in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[19] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *2012 IEEE 30th VLSI Test Symposium (VTS)*, April 2012, pp. 252–257.

[20] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "Leveraging gate-level properties to identify hardware timing channels," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, no. 9, pp. 1288–1301, 2014.

[21] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'15, New York, NY, USA, 2015, pp. 503–516.

[22] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting hardware trojans with gate-level information-flow tracking," *Computer*, vol. 49, no. 8, pp. 44–52, Aug 2016.

[23] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner, "Quantifying timing-based information flow in cryptographic hardware," in *IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15, 2015, pp. 552–559.

[24] P. Ducklin. (2013) Anatomy of a pseudorandom number generator - visualising cryptocat's buggy prng. Https://nakedsecurity.sophos.com/2013/07/09/anatomy-of-a-pseudorandom-number-generator-visualising-cryptocats-buggy-prng/.

[25] ARM, "Arm security technology - building a secure system using trustzone technology," 2009, https://developer.arm.com/technologies/trustzone.

[26] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," *Advances in Cryptology - CRYPTO'96, Springer-Verlag Lecture Notes in Computer Science*, vol. 1109, pp. 104–113, 1996.

[27] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware trojan design and implementation," *International Journal of Information Security*, vol. 10, no. 1, pp. 1–14, 2011.

[28] B. Jun and P. Kocher, "The intel random number generator," April 1999, cryptography Research, Inc. White Paper Prepared for Intel Corporation.