EasyChair Preprint

№ 11019

SoftFlow: Automated HW-SW Confidentiality Verification for Embedded Processors

Lennart M. Reimann, Jonathan Wiesner, Dominik Sisejkovic, Farhad Merchant and Rainer Leupers

October 4, 2023

SoftFlow: Automated HW-SW Confidentiality Verification for Embedded Processors

Lennart M. Reimann*, Jonathan Wiesner*, Dominik Sisejkovic†, Farhad Merchant§, and Rainer Leupers*
*RWTH Aachen University, Germany, {lennart.reimann, wiesner, leupers}@ice.rwth-aachen.de
†Corporate Research, Robert Bosch GmbH, Germany, dominik.sisejkovic@de.bosch.com
§Newcastle University, UK, farhad.merchant@newcastle.ac.uk

***Abstract*—Despite its ever-increasing impact, security is not considered as a design objective in commercial electronic design automation (EDA) tools. This results in vulnerabilities being overlooked during the software-hardware design process. Specifically, vulnerabilities that allow leakage of sensitive data might stay unnoticed by standard testing, as the leakage itself might not result in evident functional changes. Therefore, EDA tools are needed to elaborate the confidentiality of sensitive data during the design process. However, state-of-the-art implementations either solely consider the hardware or restrict the expressiveness of the security properties that must be proven. Consequently, more proficient tools are required to assist in the software and hardware design. To address this issue, we propose *SoftFlow*, an EDA tool that allows determining whether a given software exploits existing leakage paths in hardware. Based on our analysis, the leakage paths can be retained if proven not to be exploited by software. This is desirable if the removal significantly impacts the design's performance or functionality, or if the path cannot be removed as the chip is already manufactured. We demonstrate the feasibility of SoftFlow by identifying vulnerabilities in OpenSSL cryptographic C programs, and redesigning them to avoid leakage of cryptographic keys in a RISC-V architecture.**

***Index Terms*—confidentiality, property checking, information flow analysis, risc-v**

I. Introduction

Malicious modifications or unintended insecure software and hardware implementations must be detected at early design stages to avoid expensive post-silicon patches. Therefore, Electronic Design Automation (EDA) tools must consider not only performance, power, and area objectives, but also the security implications of hardware and software. Commercial approaches to running secure kernels on secured hardware have already been developed, whereby leakage paths are not considered [1]. Moreover, the academic community continues to introduce security-aware EDA tools [2].

Information Flow Analysis (IFA) gains approval when analyzing software or hardware for the leakage of secret data [3]. However, existing IFA tools do not consider the bare-metal software running on a processor when analyzing the information flow, thus giving *oversensitive* results. Leakage paths that might not be exploited by software are presented as dangerous. Although it is desirable to remove all leakage paths to untrusted components in hardware, it might not always be possible to do so. Sometimes the vulnerabilities are only detected after the manufacturing process, such as Meltdown [4]. Moreover, removing all leakage paths might result in the loss of functionality or performance. A leakage path is a signal route that carries data from a sensitive source to untrusted components (Fig. 1). Fundamentally, there are two distinct flavors of information flow analysis: dynamic and static IFA. A dynamic analysis tracks the information flow within hardware for a given software running a set of test cases [5]. As only the program flow for the test cases is considered, vulnerabilities might be overlooked. Thus, static approaches that consider software *independent* of test cases yield more reliable verification results. Most state-of-the-art tools require manual translation of either software or hardware descriptions, or limit the expressiveness of security properties. Thus, a new EDA tool is required to overcome these constraints. To address this challenge, we present *SoftFlow*, a framework that allows a *hardware-software co-verification* to analyze whether determined leakage paths are activated for a given software and arbitrary input data. SoftFlow allows the designer to adapt the program and avoid leakage, or deploy software patches for manufactured chips.

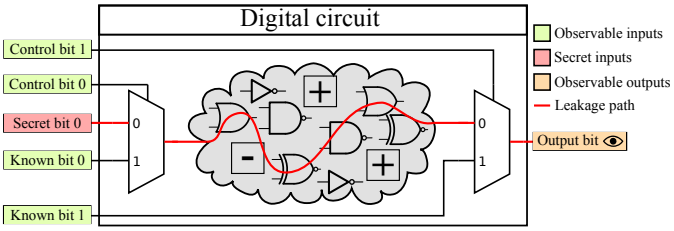


Fig. 1: A leakage path in a circuit through a multiplexer.

The contributions of this paper are: **(1)** An automated tool to convert leakage paths of a hardware description into provable hardware properties. **(2)** A complete framework that guarantees that determined leakage paths in hardware are not exploited by a given software. **(3)** An analysis of the state-of-the-art OpenSSL cryptographic algorithms to demonstrate the usability of the tool for a RISC-V architecture.

II. Related Work

As the complete system behavior can only be modeled when considering hardware and software together, a co-verification is used frequently to enforce security properties for a given design [7]. However, co-verification is a challenging task, as hardware and software are described in different domains. Typically, one description is either converted into the other domain, or both descriptions are converted to allow the verification of both implementations [8]. These methods mainly depend on model checkers that formally verify the

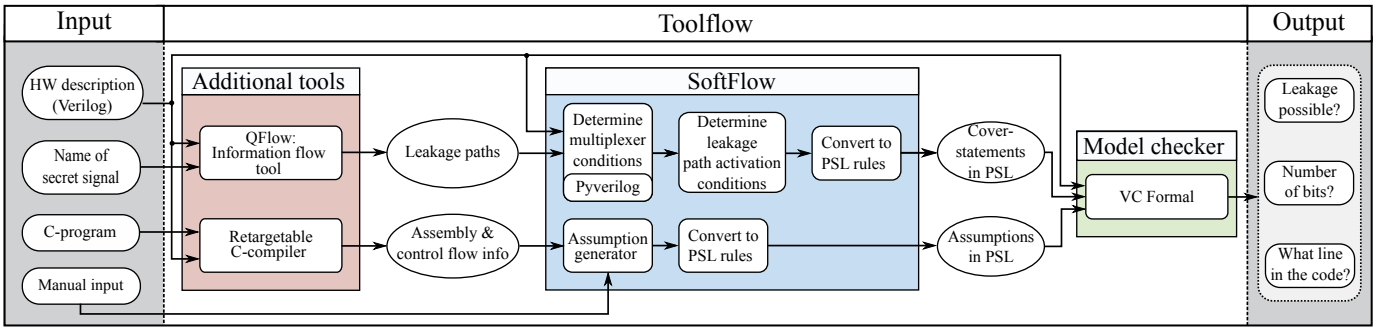


Fig. 2: Block diagram illustrating *SoftFlow's* tool flow.

defined properties. In [9], a manual translation of the hardware description, program, and properties is conducted to allow a co-verification. However, a manual translation is error-prone and could introduce vulnerabilities into the description. A model checker, similar to the one used in this work, is used in [10][11]. However, both proposals limit their use in the variability of the observed software and the expressiveness of the security property. With the introduction of SoftFlow, we overcome the mentioned limitations and establish a novel framework that allows an automated generation of security properties. Therefore, we enable a static security analysis of both hardware and software to *facilitate the design of secure software for insecure hardware*.

III. Preliminaries

A. Threat Model

The threat model is based on the following assumptions:

- The vulnerability that exposes the secret is already present at the RTL-design stage of embedded processors.
- The adversary tries to gain information about secret signals, such as cryptographic keys.
- The complete hardware design is known to the attacker.
- The adversary can observe primary inputs and outputs of the design.
- The software cannot be exchanged by the attacker.
- Arbitrary input data can be used.

B. Information Flow Analysis

IFA allows the static or dynamic analysis of programs and hardware descriptions to elaborate on whether unintended information flow can occur. Security labels are used to flag components to be part of certain *security classes* [12]. Thus, an analysis determines whether sensitive information can be leaked to untrusted areas of the hardware. For the example in Fig. 1, it is analyzed whether the secret bit can be leaked to the observable output for all possible known input bit sequences.

State-of-the-art tools can conduct a static analysis of the Verilog hardware description to determine unwanted data leakages. QFlow [13], [14] and QIF-Verilog [15] use quantitative information flow to classify data leakages as negligible or a threat. A *scopechain* is provided that allows identifying the path within the hardware description. SoftFlow utilizes QFlow to gather the most suspicious leakage path in a system.

C. Property Specification Language (PSL)

PSL [16] is compatible with Verilog hardware descriptions and allows the modeling of the temporal behavior of digital architectures. A simple example is a condition for a multiplexer: $(control\_signal == 0x20)$. The Boolean expression enables the model checker to determine when information is forwarded. Moreover, Sequential Extended Regular Expressions (SERE) can model the temporal dependencies. *CONCAT*, *FUSE*, and *REPETITION_INF* of the SERE repertoire are used in this work, as explained in Table I. The operators can be combined into sequences, allowing the modeling of a leakage path. The verification layer uses commands like `cover` and `assume`. `assume` indicates that the included property is expected to hold, thus restricting the state-space of the evaluated architecture for the desired property verification. However, the user must verify the assumptions to avoid introducing insecurities into the process. `cover`-statements can be used to determine whether a property is true for at least one state in the state-space using a model checker. If the property is proven to be uncoverable, the leakage cannot occur for the assumptions. SoftFlow automatically generates the assumptions and properties (Fig. 2).

IV. SoftFlow

For the identification of the leakage paths, SoftFlow uses QFlow, due to its capability to analyze the information flow bitwise [13]. QFlow extracts the leakage paths from the sensitive source to the possibly harmful target. The security properties are generated by SoftFlow (Fig. 2), as explained below. Additionally, model-checking requires information about

TABLE I: Sequential Extended Regular Expression (SERE) operators [6] used in this work.

Name	PSL Usage	Verilog Example	Interpretation
CONCAT	A;B	$(cntrl\_sig1 == 0x20); (cntrl\_sig2 == 0x11)$	Two cycles exist, so that first condition A is true, followed by the cycle in which B is true.
FUSE	A:B	$(cntrl\_sig1 == 0x20) : (cntrl\_sig2 == 0x11)$	A cycle exists in that both Boolean expressions A and B are true.
REPETITION_INF	A[*]	$(cntrl\_sig1 == 0x20)[*]$	The Boolean expression is true in every cycle.

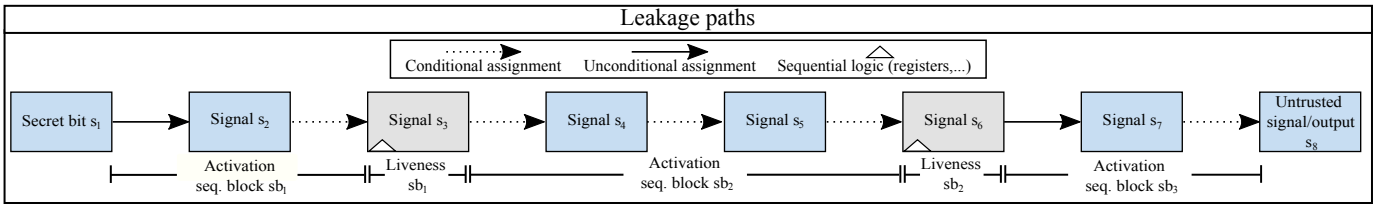


Fig. 3: Separating the leakage paths into blocks to allow a suitable environment for property generation. The arrows illustrate signal assignments in which dotted arrows describe conditional assignments.

the software to constrain the model, reducing the number of false positives. Below, we clarify how the PSL rules are generated, verified and evaluated for multiple programs.

A. Property Generation

This subsection describes a tool that processes the leakage paths, given by a static quantitative IFA, and the hardware description to generate properties using PSL.

a) Combinational: First, a combinational circuit is considered. In this case, a leakage path can consist of combinational operations and signal assignments (Fig. 1). These assignment types can be divided into two classes: conditional and unconditional. Unconditional assignments happen independently of any control signals, while conditional assignments depend on a multiplexer's control signal. When checking whether a leak can occur, one needs to analyze whether all conditional assignments are active *at the same time*. Hereafter, *active-function* A describes whether an information flow between two signals is active.

b) Sequential Blocks: Due to the timing dependency, additional system behavior needs to be considered for sequential circuits. Fig. 3 illustrates a single leakage path. The path leads from a trusted signal, storing a secret bit, to an untrusted signal. Due to the timing behavior of sequential logic within this path, each leakage path needs to be separated into multiple blocks. Assignments between non-sequential signals of the same block must occur in a single cycle. But the data can live for several cycles in sequential signals, such as registers, before moving on. Separating the leakage paths into blocks that always end with a sequential signal allows for a differentiation of this behavior. The separation into the sequential blocks (sb) is depicted in Fig. 3. Additionally, one must consider that the secret data is not overwritten within the sequential logic before forwarding it to the next sb.

Only conditional assignments influence the property generation. For the first sequential block in Fig. 3, only the second assignment holds a condition. Hence, the *block active-function* BA for this sequential block sb_1 is true if the conditions for the assignment are true in a single cycle.

1) Leakage Path Activation: First, the sequential logic within the leakage path is identified. This is done to form the sequential blocks. The secret datum can only pass a single sequential block in each cycle. For leakage to occur, the second sequential block needs to be activated after the previous one (Fig. 3). Additionally, the two blocks do not have to be activated in consecutive cycles as long as the information in the previous sequential logic stays alive and is not overwritten until it passes the next sequential block, which is described with the *alive-function* L. Each sequential block results in a BA and L function, except for the last one. In the example (Fig. 3), two alive-functions and three block active-functions are required to describe the activation of the path.

2) Active & Alive-Function Collection: First of all, the Verilog description is parsed to form a tree structure that can be iterated over. Afterward, the algorithm iterates over the leakage paths to determine the individual activation conditions. The conditions for a single signal assignment are connected via an OR-operator as only a single true activation condition can lead to leakage. The resulting set of active-functions are temporally stored in a file to be processed to PSL. The alive-function L is determined similarly to the active-function. The alive-function returns 'true' if the current value in a sequential logic is not overwritten for a fixed amount of cycles. An empty signal assignment describes a value that is not overwritten. The activation condition from the empty node to the destination node is calculated and stored as the alive function.

3) Specifying SoftFlow's PSL Sequences: The collected active-functions and alive functions need to be translated to PSL to enable model-checking. For the example in Fig. 3, the PSL sequence is shown in Equation 1, with AS and LS describing the activation and liveness sequences. The sequence is extended with the `cover` command, forming the *cover rules*. If the model checker can cover this path, the leakage can occur for the given assumptions. The individual Boolean activation conditions are connected using the *FUSE* operator (Table I), as the conditions must be true in the same cycle. After the activation conditions are converted, the liveness condition for the sequential logic is required. The returned liveness condition is converted to a PSL expression using the *REPETITION_INF* operator (Table I). The activation conditions and the alive-functions are combined into the full sequence for a single sb using *CONCAT* (Table I). *Uncoverable properties state that the leakage path cannot be taken.*

$$\underbrace{\underbrace{\underbrace{\textbf{active}(s_2,\, s_3)}_{AS}\ ;\ \underbrace{\textbf{alive}(s_3)[*]}_{LS}}_{sb_1}\ ;\ \underbrace{\underbrace{\textbf{active}(s_3,\, s_4) : \textbf{active}(s_4,\, s_5) : \textbf{active}(s_5,\, s_6)}_{AS}\ ;\ \underbrace{\textbf{alive}(s_6)[*]}_{LS}}_{sb_2}\ ;\ \underbrace{\underbrace{\textbf{active}(s_7,\, s_8)}_{AS}}_{sb_3}} \tag{1}$$

B. Property Verification

By employing model-checking, one can establish a level of *certainty* regarding the reliability of the verification results. The individual cover rules allow an independent evaluation for every leakage path, enabling the model checker to work in parallel for these properties. The program memory address is used to identify the instruction and the related C code line.

SoftFlow's generated properties do not consider the compiler program for this processor. As we would like to elaborate on the security of a given C program and an insecure hardware implementation, the hardware model for the model-checking needs to be constrained before running the verification. This is done to reduce the false positives, which describe a leakage of secret data for programs that would not be implemented. Those constraints are implemented using `assume`. Since the model permits the data memory to contain any information, it is necessary to contemplate all conceivable routes within a program. Conditional branches depend mostly on data in the data memory. Thus, all possible conditional jumps need to be considered. The user can choose from the following derived assumptions (Ⓐ to Ⓕ) to restrict the program flow.

Ⓐ *No illegal instructions:* A assumption is made which claims that the read port of the program memory can never hold the value of prohibited instructions.

Ⓑ *Only used instructions:* The formal model can be further constrained by analyzing the compiled machine-code statically. An assumption can be generated that further constrains the model by stating that only instructions present in this compiled program can be read from the program memory. The order is not yet considered.

Ⓒ *Replacing the program memory:* To avoid modeling an entire program memory, a lookup table is used instead, which also enables consideration of the order of instructions. Furthermore, since return addresses of function calls are stored on the stack, it is possible for a return to transpire to any point within the program.

Ⓓ *Only legal return addresses:* The remaining three assumptions are generated using compiler information. The assumption is used to remove undefined return addresses, which results in arbitrary program flows for the verification.

Ⓔ *Only correct hardware jumps:* Compiler information is processed to allow only valid start and return addresses for hardware loops during the evaluation.

Ⓕ *Call-return matching:* When a function is invoked from several locations, it is essential to account for both return addresses. Confirming a legitimate correlation between a call and its corresponding return is only achievable during runtime. Nonetheless, if an authentic hardware call-stack is utilized to store the return addresses, it can already be taken into consideration during the verification process.

V. Evaluation

The evaluation is conducted for a RISC-V Verilog description and compiler. The processor uses the RV32IC instruction-set, in which an instruction for a single round of AES, and an additional memory have been added that holds the cryptographic key. A block diagram illustrating the architecture is shown in Fig. 4. The data input from the key memory is labeled *sensitive* in order for QFlow to yield the most suspicious leakage paths from the signal to all output ports of the design, including the memories' data and address ports. The analysis is performed for several cryptographic algorithms from OpenSSL[17]: ChaCha20, AES-256, Camelia, Aria, and SHA-256. SoftFlow is evaluated for different assumptions to elaborate on their efficiency and security.

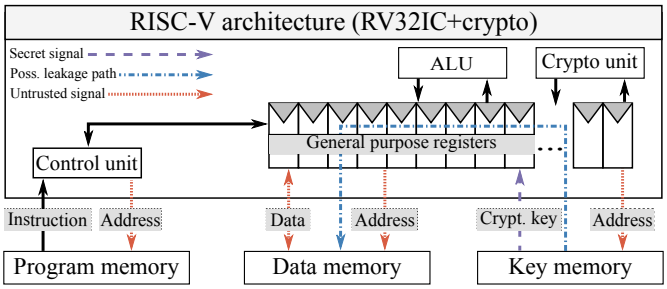


Fig. 4: Abstract block diagram of the RISC-V architecture.

A. Verification Cases

The used assumptions are listed behind each mode.

NONE: No assumptions are added to the model checker. The verification tool works solely with the properties generated from our property generator for the leakage paths.

LEGAL Ⓐ: The loading port of the program memory is restricted, disallowing all illegal instructions to be read.

USED Ⓐ & Ⓑ: The data port of the program memory is restricted using an `assume` statement that allows only instructions present in a compiled machine-code to be read.

JUMPS Ⓐ to Ⓔ: In this mode, only valid return addresses can be used for a return command and hardware loops, allowing a more realistic program flow.

STACK Ⓐ to Ⓕ: A hardware stack allowing the return-call matching is added as supplemental Verilog code.

FULL: First, the assumptions for USED are utilized. If the property for this path cannot be proven to be uncoverable, all verification cases are tried until the model checker is successful. Considering the valid states S of a processor model, the valid states for STACK are present in USED, resulting in:

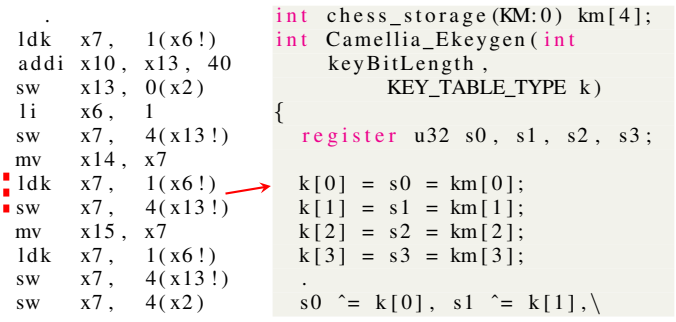
$$S_{\text{STACK}} \subseteq S_{\text{JUMPS}} \subseteq S_{\text{USED}} \subseteq S_{\text{LEGAL}} \subseteq S_{\text{NONE}}$$

The inclusiveness of the more restrictive verification cases further reduces the state space that has to be analyzed.

B. Results

1) Coverages: The evaluation results of SoftFlow's functionality are presented below. The verification modes are applied to the five cryptographic algorithms for the RISC-V.

a) Without Software:: The assumptions of the modes NONE and LEGAL are applied to the verification model. These two modes are independent of the actual software running. QFlow yields **3776** leakage paths that need to be elaborated for the given architecture. Leakage paths are **uncoverable** if



(a) Assembly program (b) C program

Fig. 5: A naive implementation of Camellia: The dashed line symbolizes the reported instructions activating the leakage path. The corresponding C-code is marked with an arrow.

the leakage cannot occur for the given assumptions. The paths that can be activated are labeled **covered**. Both, the LEGAL and NONE modes yield **857** uncoverable paths, which indicates that the processor only accepts legal instructions and that the 857 paths are false positives from QFlow. *With SoftFlow, the false positives are removed entirely!*

b) With OpenSSL Cryptographic Algorithms:: Next, the capabilities of SoftFlow are elaborated for the different OpenSSL algorithms for the remaining verification modes. An example of how the applications are modified for the architecture is shown in Fig. 5b. The key can be accessed with a pointer to the key memory KM. Table II illustrates the results of the elaboration for the modes STACK, JUMPS, and USED. As one can see, the number of uncoverable paths increases for all applications when more restrictive assumptions are used. No difference in the results between the modes STACK and JUMPS can be observed.

The call tree depths are limited for all five applications, which is required if the STACK mode is used. Otherwise, the hardware stack would have to be of unlimited size. Thirty-two leakage paths are still covered for the design and applications. The cause for this leakage detection can be seen in Fig. 5 for the application Camellia. The remaining exemplary elaborations are only presented for Camellia. Final results are presented for all applications. The assembly code in Fig. 5a shows that the compiler loads the key values from

TABLE II: Summarized metrics for the five applications grouped by verification case.

	Application	Aria	ChaCha20	Camellia	AES	SHA
	Instr. count	2112	1347	1973	1409	649
	Call depth	4	4	4	6	5
USED	Covered	65	74	58	58	48
	Uncoverable	3711	3702	3718	3718	3728
JUMPS	Covered	32	32	32	32	32
	Uncoverable	3744	3744	3744	3744	3744
STACK	Covered	32	32	32	32	32
	Uncoverable	3744	3744	3744	3744	3744

```
14    s0 ^= km[0], s1 ^= km[1],\
15    s2 ^= km[2], s3 ^= km[3];
```

Listing 1: Replacing k with km in the application camellia

```
1    volatile int chess_storage(KM:0) km[4];
```

Listing 2: Declaring km as **volatile**

```
1  if (input[0] == 0x4d2)
2  {
3    u32* leakage = (u32*) malloc(sizeof(u32)*8);
4    for (int j=0; j<8; j++) leakage[j] = km[j];
5  }
```

Listing 3: The malicious lines implementing a simple Trojan in the application Camellia.

the dedicated key memory (*ldk*, load key) and stores them in the untrusted data memory (*sw*, store word) for the array *k*. The programmer can avoid the usage of the leakage path by modifying the C program. Listing 1 shows the replacement of the variable *k* with the pointer *km*, so that the values are not automatically stored in the untrusted data memory. Additionally, the pointer is marked as *volatile* (Listing 2), instructing the compiler not to optimize any data movements for KM's data. Table III depicts the outcomes of SoftFlow subsequent to this modification. The leakage paths are not activated anymore except for the Aria application. For Aria, the compiler places the key in a register that is directly connected to the address port of the key memory, which is marked as untrusted. Forcing the compiler to pick a different register can also avoid this leakage. However, the limited intrusion into the compiler does not allow this change. The final evaluation shows an example of finding software Trojans inside the program. In the four applications that achieved 100% avoidance of leakage, a software Trojan is implemented. An example Trojan for the Camellia application can be seen in Listing 3. For a particular input value stored in the data memory, the keys are written to the untrusted memory.

The results for the evaluations of the FULL mode illustrate *that all Trojans that use the leakage paths are detected despite their difference in the trigger*, as shown in Table IV. The value of the trigger does not play a role in the detection, as the data coming from the untrusted data memory is assumed to be of any possible value.

2) Runtime: For the unmodified applications (e.g., Fig. 5b) that still carry 32 covered paths, the verification is conducted by using the available verification modes. LEGAL is not presented here, as it yielded the same verification results as NONE. As shown in Fig. 6, the lowest runtimes are given by the verification modes that restrict the hardware model the least. However, some leakage paths could not be marked as **uncoverable** for the less restrictive assumptions (Table II).

TABLE III: Metrics of the modified applications.

	Application	Aria	ChaCha20	Camellia	AES	SHA
FULL	covered	473	0	0	0	0
	uncoverable	3303	3776	3776	3776	3776
	time	16016	16780	3213	1337	4432

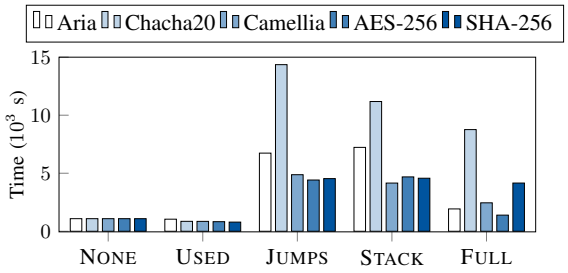


Fig. 6: The summarized runtime of the verification procedure for the different verification modes and applications.

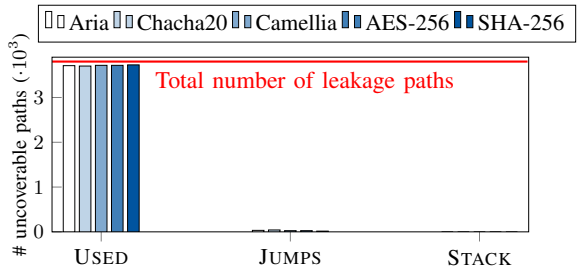


Fig. 7: A histogram showing what modes are mostly successful when verifying the model for different applications.

The runtime increases drastically for all applications at JUMPS. Moreover, as expected, the lowest runtime can be achieved using the verification mode FULL. The overall runtime is optimized, while yielding a precise result for all applications. As the verification of the properties for every leakage path can be parallelized, depending on the available resources of the designer, the runtimes can be drastically reduced. Fig. 7 illustrates how successful the different verification modes are in their task. It can be observed that most leakage paths can be flagged as uncoverable by assuming that only the instructions given in a program will be used. The higher number of cases verified using USED explains the reduced runtime of the FULL-mode in Fig. 6.

VI. LIMITATIONS

Overall, the tool can identify vulnerabilities and assist the designer in writing software that allows safe use of insecure hardware. However, some limitations must be considered.

Leakage paths can be triggered with instructions like load and store operations, as shown in Fig. 5. The data is leaked without any change. This results in a varied attack scenario from the one used by QFlow. Additionally, the evaluation is only conducted on 3776 existing leakage paths, as QFlow only outputs the most suspicious ones. Although the elaboration of the paths was successful, all leakage paths must be considered. It might not be possible for some applications and hardware combinations to avoid leakage. This could be due to unconditional leakages or leakages caused by instruction patterns required for the application.

VII. CONCLUSION

We presented SoftFlow, an EDA tool that utilizes formal verification to conclusively indicate whether leakage paths in hardware are activated for a given software. Therefore, SoftFlow facilitates both a security-driven system design from the ground up and post-fabrication security patches. The usability of the proposed framework was demonstrated by analyzing OpenSSL cryptographic algorithms for a RISC-V architecture. The evaluation proves that secure software disabled all 3776 leakage paths. In addition, the consideration of both software and hardware mitigated all false positives of QFlow. With SoftFlow, we enable a security-aware software-hardware co-verification process that takes into account the intricate interplay of dedicated hardware and software. In the future, we plan to investigate security-aware compilers that facilitate the complete removal of vulnerabilities if allowed by both hardware and software.

TABLE IV: Verification of the modified applications with embedded Trojans.

	Application	ChaCha20	Camellia	AES	SHA
FULL	covered	32	32	32	32
	uncoverable	3744	3744	3744	3744
	time	26045	2127	1386	1321
	trigger (hex)	12345678	4d2	fedcba	12ef34dc

REFERENCES

[1] D. Sisejkovic *et al.*, "A Secure Hardware-Software Solution Based on RISC-V, Logic Locking and Microkernel," ser. ACM SCOPES '20.

[2] IEEE Council on Electronic Design Automation, "CADForAssurance," 2020. [Online]. Available: https://cadforassurance.org/

[3] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *DATE '2017*.

[4] M. Lipp *et al.*, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore.

[5] O. Ruwase *et al.*, "Parallelizing dynamic information flow tracking," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 01 2008, pp. 35–45.

[6] A. Cimatti *et al.*, "From Sequential Extended Regular Expressions to NFA with Symbolic Labels," in *Implementation and Application of Automata*. Springer Berlin Heidelberg, 2011.

[7] F. Lugou *et al.*, "Toward a Methodology for Unified Verification of Hardware/Software Co-designs," *Journal of Cryptographic Engineering*, pp. 1–12, Nov. 2016.

[8] B.-Y. Huang *et al.*, "Formal Security Verification of Concurrent Firmware in SoCs using Instruction-Level Abstraction for Hardware," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*.

[9] X. Guo *et al.*, "Scalable SoC trust verification using integrated theorem proving and model checking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2016, pp. 124–129.

[10] M. D. Nguyen *et al.*, "Formal hardware/software co-verification by interval property checking with abstraction," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 510–515.

[11] D. Große *et al.*, "HW/SW Co-Verification of Embedded Systems Using Bounded Model Checking," ser. GLSVLSI '06.

[12] J. Oberg *et al.*, "Theoretical analysis of gate level information flow tracking," in *Design Automation Conference*, 2010, pp. 244–247.

[13] L. M. Reimann *et al.*, "QFlow: Quantitative Information Flow for Security-Aware Hardware Design in Verilog," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 603–607.

[14] L. M. Reimann *et al.*, "Quantitative information flow for hardware: Advancing the attack landscape," in *2023 IEEE 14th Latin America Symposium on Circuits and System (LASCAS)*, 2023.

[15] X. Guo *et al.*, "QIF-Verilog: Quantitative Information-Flow based Hardware Description Languages for Pre-Silicon Security Assessment," in *IEEE HOST '19*.

[16] C. Eisner *et al.*, *A Practical Introduction to PSL Introduction*, ser. Integrated Circuits and Systems. Springer, Boston, MA, 2006.

[17] Open SSL Project, "OpenSSL," 1998. [Online]. Available: https://github.com/openssl/openssl