Security Verification of Low-Trust Architectures

Qinhan Tan*
qinhant@princeton.edu
Princeton University
Princeton, New Jersey, USA

Yonathan Fisseha*
yonathan@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Shibo Chen*
chshibo@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Lauren Biernacki
biernacl@lafayette.edu
Lafayette College
Easton, Pennsylvania, USA

Jean-Baptiste Jeannin
jeannin@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Sharad Malik
sharad@princeton.edu
Princeton University
Princeton, New Jersey, USA

Todd Austin
austin@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

ABSTRACT

Low-trust architectures work on, from the viewpoint of software, always-encrypted data, and significantly reduce the amount of hardware trust to a small software-free enclave component. In this paper, we perform a complete formal verification of a specific low-trust architecture, the Sequestered Encryption (SE) architecture, to show that the design is secure against direct data disclosures and digital side channels for all possible programs. We first define the security requirements of the ISA of SE low-trust architecture. Looking upwards, this ISA serves as an abstraction of the hardware for the software, and is used to show how any program comprising these instructions cannot leak information, including through digital side channels. Looking downwards this ISA is a specification for the hardware, and is used to define the proof obligations for any RTL implementation arising from the ISA-level security requirements. These cover both functional and digital side-channel leakage. Next, we show how these proof obligations can be successfully discharged using commercial formal verification tools. We demonstrate the efficacy of our RTL security verification technique for seven different correct and buggy implementations of the SE architecture.

CCS CONCEPTS

• **Security and privacy** → **Formal methods and theory of security**; **Information flow control**; **Security requirements**; *Side-channel analysis and countermeasures.*

KEYWORDS

Low-Trust Architecture, Information Flow, Formal Verification

*Three co-first authors contributed equally to this research.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616643

ACM Reference Format:
Qinhan Tan, Yonathan Fisseha, Shibo Chen, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. 2023. Security Verification of Low-Trust Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616643

1 INTRODUCTION

Security verification of a computing system, while highly desirable, is a challenging task that often falls short of the desired level of guarantees. The verification must be applied to all trusted components of the system, including hardware and software. Unlike penetration testing, which consists of focused attempts to infiltrate a system, formal security verification is a proof that a particular security vulnerability does not exist within a design. Unfortunately, most systems today receive little to no formal security verification, due to design complexity challenges and limitations of formal proof mechanisms. Design complexity manifests in the sheer size of today's secure systems, which comprise architectures, microarchitectures, and deep software stacks, all of which must be trusted and verified. These complex systems easily exceed the capabilities of today's formal proof mechanisms, such as SAT/SMT solvers, model checkers, and proof assistants. Consequently, incomplete penetration testing still remains the backbone of today's security verification efforts.

Low-trust architectures have recently emerged as a secure system design framework that *i)* eliminates all trust in software, and *ii)* significantly reduces the amount of hardware trust to a small, software-free enclave component. These properties make formal security verification feasible by shrinking the system aspects that must be trusted and verified. In this paper, we focus on the security verification of Sequestered Encryption (SE) [13]—a low-trust architecture that claims to protect the confidentiality of sensitive data against direct data disclosures and digital side channels. Direct disclosures refer to any direct leakage of plaintext values through SE computation. Digital side channels represent any indirect leakage of plaintext values through non-analog information paths, including

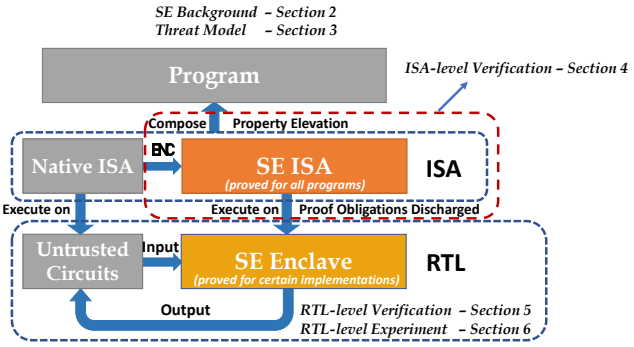

Figure 1: Proof System and Paper Organization

analysis of ciphertext values, operational timing, program control flow, memory access patterns, or microarchitectural resource usage. Currently, SE Enclave does not protect against analog information flow paths, such as frequency throttling [53, 77], power analysis [34, 49, 56], electromagnetic snooping [2, 63, 70], etc.

Within SE, the instruction set architecture (ISA) consists of the native instructions, termed the native ISA, and a set of secure instructions termed the SE ISA. The native ISA contains insecure instructions to be executed by unsecured (or untrusted) components that do not have access to secret values. In contrast, the SE ISA consists of secure instructions that operate on encrypted data and are executed solely in its software-free enclave component. It is the design of this SE Enclave and ISA extension that ensures the cryptographic-strength confidentiality of the sensitive data.

In this work, we perform a complete formal verification of the SE low-trust architecture to show that the design is secure against direct data disclosures and digital side channels for all possible programs. The steps involved are illustrated in Figure 1. First, we articulate a set of instruction-level security requirements that SE ISA must fulfill to prevent disclosures or digital side channels. Going upwards to the software, these are used in hand-driven proof techniques to show that no SE program can possibly create a disclosure or digital side-channel leakage. Going downwards toward the hardware, the ISA security requirements necessitate security properties to be enforced in the hardware design, typically called RTL-level security properties. Last, we propose a verification scheme to formally verify RTL-level properties using an off-the-shelf commercial formal verification tool (Cadence's JasperGold [19]) on real designs. Our evaluation experiments show that our RTL-level verification scheme can prove the security properties being met by correct implementations, and also capture security leakages in flawed implementations. This demonstrates the practical applicability of our proposed verification scheme to designs at scale. To our knowledge, this is the first formal verification of a secure computing framework that extends to both direct disclosures and digital side channels, for all possible programs running on a verified computing platform.

A key takeaway from this work is that low-trust architectures lend themselves to formal security verification. We find two primary reasons for this outcome. First, the nature of low-trust architectures eliminates any trust in software. Since our verification ensures that the software can only see values encrypted under semantically secure cryptosystems [33], software verification is not part of the overall proof system. Using only hand-driven proof, we show that any program using our ISA cannot disclose or create digital side channels, thus ending concern for any software. In traditional security verification, where properties must be proven partly in hardware and software, often the complexity of software reasoning leads to compromises on what can be guaranteed in these systems. Second, the simplicity of the low-trust SE hardware enclave, having minimal state and control logic, allows all of our ISA-level-based security assertions to complete on the actual RTL of the design, ensuring no gaps between the deployed design and any potential abstractions employed to enable formal verification. We are confident that the approach we have detailed in this paper will extend itself to future low-trust architectures as they become available.

Contributions. This work observes and demonstrates how low-trust architectures enable end-to-end software-to-hardware verification of strong security attributes, *i.e.*, confidentiality and digital side-channel free execution. Below is the list of specific contributions:

- **Formal SE ISA Semantics**: Define formal SE ISA semantics that enable privacy-related reasoning.
- **Software-Level Proof**: Povide proof of confidentiality and side-channel freedom for ***all programs*** written using this ISA.
- **Hardware Proof Obligations**: Provide proof obligations for any hardware implementation of the SE ISA to serve as the interface between the verification of the software and the hardware.
- **Hardware-Level Proof**:
  - Provide a list of security properties that meet the hardware proof obligations for a specific SE hardware implementation.
  - Demonstrate that these properties can be checked using standard information flow tracking (IFT) and commercial off-the-shelf IFT tools with novel RTL-verification elements.
  - Demonstrate how the checking detects bugs in four buggy different implementations that violate these properties.

To further clarify the contributions of this work, we do not claim generalizability beyond the small enclaves in low-trust architectures detailed in §2. In fact, the increased level of verification is enabled by the low-trust architecture's property of limiting trust to only within the small SE Enclave, which in turn eliminates all trust in software and significantly reduces the degree of hardware that must be trusted (and thus needs to be verified) to ensure the proof properties.

Figure 1 also serves to illustrate our paper organization. In § 2, we provide a brief overview of SE. In § 3, we articulate the threat model and the scope of this work. In § 4, we formalize ISA-level properties and prove the program-level properties inducted from the ISA-level properties followed by proof responsibilities discharged to the RTL-level. In § 5, we present our modeling and verification strategy at the RTL-level. In § 6, we describe the SE design variances and apply our verification scheme to these designs. We close the paper with related works (§ 7) and final conclusions (§ 8).

Our designs and verification scripts are available at https://github.com/qinhant/SE_verification_CCS.

2 BACKGROUND

In this section, we provide an overview of Sequestered Encryption (SE) [13] and show that the unique characteristics and design principles of SE open up new opportunities for hardware verification to provide complete reasoning of the underlying computing paradigm

without the knowledge of specific programs. A more detailed study of the SE design, including comparisons with related work, can be found in [13].

2.1 Sequestered Encryption (SE)

Sequestered Encryption (SE) is a hardware-based technique to protect the confidentiality of secret third-party data during computation. With SE, third-party data is encrypted in a trusted, client-side environment and offloaded to a server for computation. The server application operates on third-party data using SE's ISA instructions backed by custom hardware support. Specifically, SE extends the conventional ISA to include secure instructions that operate on ciphertext data. We call these additional secure instructions the SE ISA. These instructions are dispatched to SE's hardware enclave, which computes the requested operation on the source ciphertexts.

The SE Enclave works to significantly reduce software and hardware trust by sequestering all sensitive computation to the hardware enclave. As such, the SE Enclave design claims that sensitive private data cannot be disclosed by any SE instruction sequence, either directly through a disclosure or indirectly through a digital side channel. This claim covers all digital side channels, including cryptanalysis of the ciphertext emitted by the Enclave, Enclave operational timing, and any possible influence the SE Enclave has on the system's memory access patterns and control flow. The goal of this paper is to formalize and prove these claims for the SE Enclave, through proofs on the SE instruction set operational semantics and their proof obligations expressed in the RTL implementation in the SE Enclave. ***With these claims proven, the SE Enclave represents the first enclave that has been formally verified to not suffer from software vulnerabilities or digital side channels.***

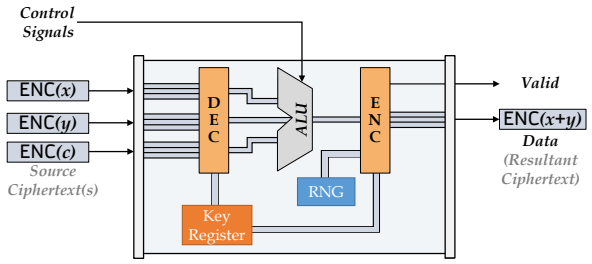

Figure 2: SE Enclave Design

2.2 SE Instruction Set Architecture (ISA)

SE extends the native ISA with ***secure instructions*** that explicitly operate on ciphertext data. When the processor decodes a secure instruction from the SE ISA, the instruction is dispatched to the SE Enclave for processing. This operation mirrors how native instructions are dispatched to functional units in a conventional processor. Insecure instructions in the native ISA are dispatched to unsecured functional units. SE makes no claim about these instructions, as they are executed in an untrusted environment.

To ensure that sensitive data does not leave the enclave, SE restricts its secure instructions to be 'data-oblivious,' only supporting arithmetic, logical, comparison, and shift operators. The classes of instructions supported by the SE ISA are summarized in Table 1. Specifically, the SE ISA does not support control flow or memory instructions, as these would innately leak information about sensitive data through architectural states like the program counter. Insecure versions of these instructions (*i.e.*, those assumed to be operating on public data) are still present in the native ISA. SE enables secure control flow via an encrypted conditional move (`CMOV`) instruction. Secure `CMOV` instructions function as ternary operators, where a destination register is updated based on the value of some condition, akin to predicated instructions. This primitive enables programmers to make decisions on secret conditions, mimicking the logic of if-statements, in a safe manner. Finally, to perform plaintext-ciphertext operations, the processor must first encrypt the plaintext value using SE's `ENC` instruction before passing the resultant value to another secure instruction for computation.

Instruction Class	Example		Secure (SE Enclave)	Insecure (Native)
Encryption	`ENC`	Encryption	●	○
Arithmetic	`ADD`	Addition	●	●
Logical	`AND`	Logical And	●	●
Comparison	`LT`	Less Than	●	●
Shift	`SLL`	Logical Left Shift	●	●
Conditional	`CMOV`	Conditional Move	●	●
Memory	`LD`	Load	○	●
Control Flow	`JMP`	Jump	○	●

Table 1: Summary of ISA

2.3 SE Enclave Implementation

The SE Enclave is architected as a small hardware functional unit embedded within the execute stage of the pipeline. This hardware unit includes storage of the secret key under which the user data is encrypted. When instructions are dispatched to the SE Enclave, the unit decrypts ciphertext source operands under this key, computes the requested operation in plaintext, then re-encrypts the result. Namely, the syntax for a secure `ADD` instruction is `ENC( DEC(r1) ADD DEC(r2) )` This operation is illustrated in Figure 2. The resultant ciphertext value is the only value that leaves the enclave. The SE Enclave implementation is trusted and assumed to be free of direct data leakage. Further, the SE Enclave is implemented to have data-independent timing, such that timing the execution of the enclave cannot reveal the plaintext values of ciphertexts. For example, the SE Enclave cannot contain hardware optimizations that accelerate instructions for specific inputs (*e.g.*, forwarding for multiply-by-zero [35]), as these optimizations leak information via timing.

While the design in Figure 2 implements the syntax of the SE ISA, the SE Enclave can have many instantiations, including different implementations of encryption and decryption. Below, we discuss some of the different variations of SE Enclave presented in [13]. In this work, we present different RTL instantiations in Section 6.

2.3.1 Encryption Scheme. In SE, a value (m) is encrypted by appending a fresh random salt (u), then applying a strong pseudorandom permutation (PRP), such as a block cipher. In this encryption scheme, both the data value and salt are 64 bits long, thereby producing a 128-bit ciphertext. This scheme can be implemented in hardware with a variety of symmetric or asymmetric encryption ciphers built as rolled or unrolled implementations. Prior work analyses three cryptographic ciphers for use within SE: AES-128, QARMA, and Simon. In this work, we also consider the popular asymmetric public-private key cryptosystem RSA.

The SE encryption scheme salts each ciphertext value with fresh randomness through the True Random Number Generator (TRNG) to ensure that ciphertexts are diversified to thwart cryptanalysis attacks and some side-channel attacks, *i.e.*, CIPHERLEAKS [52]. This requirement is specified in the ISA and can be ensured by a structural RTL check.

2.3.2 Optimizations. SE can also be implemented with internal optimizations that seek to improve its performance. For example, [13] proposes caching recently decrypted ciphertexts in order to bypass the decryptor for instructions with data dependencies. Optimizations have the potential to invalidate the security claims of SE. In this work, we assess several secure and insecure optimizations of the SE Enclave to demonstrate that the secure optimizations are fully verified and the insecure optimizations are detectable by our novel verification technique.

2.4 Opportunities for Hardware Verification

SE's security claims are based on the assumption that the hardware is secure. The SE architecture positions itself well for formal verification because it has a minimal hardware footprint and possesses no trusted software, which is in sharp contrast to other Trusted Execution Environments (TEEs), like Intel SGX. As noted by prior work [22], formally reasoning about existing enclaves including Intel SGX is infeasible as any proof would have to model all processor features that exposed registers. Further, such work would be short-lived as any architectural modifications would invalidate this security proof. Rather, SE's compact design allows it to be verified independently of other structures of the processor, thus avoiding the deficiency of verifying the whole processor. In this work, we formally verify SE's security claims to establish trust in the SE ISA and its implementation.

3 THREAT MODEL

In this section, we describe and discuss our threat model. We first present our security goals. This is followed by listing the attacker's capabilities which can compromise these goals. Finally, we describe the root of trust which specified the components that can be assumed to be trusted.

3.1 Security Goals

In this work, we will formally verify that SE ISA and SE Enclave RTL implementations preserve data confidentiality. Specifically:

- Any program should not leak sensitive data through architectural states and/or side channels through instructions from the SE ISA.
- Any SE secure instruction or sequence of instructions should not leak sensitive data through microarchitectural states and IO signals of the SE Enclave.

3.2 Attacker Capabilities

In this work, the attacker's goal is to, within a reasonable amount of time, infer the plaintext secret values by analyzing the program's computation results, snooping on SE Enclave's IO signals, or analyzing the program execution time. We consider attackers to possess the following capabilities:

- The attacker can observe and/or arbitrarily change digital signals outside of the SE Enclave including signals on SE Enclave IO. These signals can be observed at every cycle even though each cycle may not result in an architectural state update.
- The attacker can run any program using secure/insecure instructions, and measure the program execution time.
- The attacker **cannot** measure and/or arbitrarily change the states inside the SE Enclave and the physical characteristics of the chip.

3.3 Root of Trust

This work assumes the following components and algorithms, which have been studied extensively in prior works, can be trusted and meet design requirements:

- *True Random Number Generator (TRNG).* We assume there exist TRNG designs that are capable of supplying at least s (for example, $s = 64$ in our designs) bits of random number per cycle. Prior works have proposed TRNGs with different designs [11, 24, 37, 50, 62, 66, 81, 86]. Recent laser-based random number generators can generate up to 250 terabits per second [47]. This approach is also shown to be bias-free in nature[78].
- *Key Exchange Mechanism.* We assume there exists a safe key exchange mechanism for the user and SE Enclave to establish shared keys. Key exchange methods like RSA key exchange, Diffie-Hellman (DH), and Elliptic-curve Diffie-Hellman (ECDH) have been well-studied [17, 40, 59, 65]. Public key infrastructure (PKI)[15, 42, 57, 79] has been developed over the years to provide authenticity guarantees. In SE, key exchange can be done using a standard small-footprint hardware-only implementation in the Enclave, as in typical Hardware Security Modules [8], which puts the key directly into the key register and does not interact with the rest of the Enclave. The only output of the key register is shown in Figure 2 and our verification proves the key is secure *once in the key register*. Verifying key exchange is orthogonal to this work.
- *Encryption Scheme.* We assume popular encryption algorithms are strong and robust against crypto-analysis when paired with long enough keys. Multiple hardware-friendly encryption schemes have been proposed and can be used in SE, *e.g.*, AES-128[38, 41], Simon-128/128[12], QARMA$_{11}$-128-$_{\sigma 1}$[9].

A more detailed account of RTL-level assumptions that flow from the above threat model will be discussed in § 5.1.

4 SE ISA MODELING AND ANALYSIS

A conventional ISA does not place any requirements on microarchitectural states and state transitions as it only requires functional correctness. However, the SE ISA needs to make explicit and verifiable rules such that the system can guard off software-based attacks. On the hardware side, the microarchitectural implementations need to faithfully follow the design requirements set by the ISA in a verifiable manner. The security properties resulting from these rules should be attested with formal proofs on the implementation side.

From the perspective of a program defined in such an ISA, there are two types of data: user ***private data*** and ***public data***. Private data are sensitive data that are always encrypted under secured keys. The value of private data should only be visible to trusted and formally verified hardware components and remain invisible to any software or untrusted components. Public data are non-sensitive data that are stored in plaintext, *i.e.*, program code, public constants,

etc. While the conventional ISA is sufficient to operate on public data, a specialized set of instructions needs to be implemented in order to process private data securely. This set of secure instructions specifies additional requirements, which are not needed for public data only programs, to securely execute programs that compute on secured sensitive data. While previous work [13] has described these properties informally, we give precise definitions of the properties in this work and state proof obligations for the RTL verification to ensure that the implementation is faithful to the ISA specification. These requirements can be organized into two groups: ***Direct Disclosure Safety*** and ***Indirect Disclosure Safety***. We will now consider them one at a time in the following sections.

4.1 ISA Direct Disclosure Safety Requirements

SE ISA must define the trust boundary between secure instructions and insecure instructions. This boundary is then guaranteed by the architecture. Inappropriate disclosure at the ISA level can happen in two ways: first, instructions could directly disclose the private plaintext data to insecure locations (*e.g.*, registers that are readable by the attacker); second, the ciphertext is produced by a weak encryption cipher, making it vulnerable to cryptanalysis attacks. Therefore, for any valid SE program, which is a sequence of instructions defined by the secure ISA, we have the following two requirements. First, we put the obvious restriction that there are no direct disclosures; and second, we put a requirement on the quality of the ciphertexts produced by the secure instructions.

4.1.1 Direct Information Disclosures. SE instructions take either a ciphertext or a public plaintext and produce a ciphertext as a result. Any secret data must never be disclosed to insecure (*i.e.*, non-SE) instructions. This is reflected at both architectural and microarchitectural levels since SE uses encryption to hide sensitive information from untrusted software and hardware. The encryption module marks the trust boundary between the hardware components since all data is re-encrypted before it is emitted out of the trusted execution component and exposed to untrusted software and hardware. Encrypted data can be safely stored in untrusted storage or computed on by an insecure instruction.

The instruction level properties lift to programs naturally. All SE instructions produce ciphertext and there is no decrypt instruction in the ISA. Thus, non-SE instructions cannot get access to the plaintext. From this, we conclude that any composition of SE instructions and non-SE instructions cannot disclose the secret data.

4.1.2 Quality of Ciphertexts. Ciphertexts need to be safe from cryptanalysis when disclosed to an attacker of reasonable strength. In this work, we consider security against chosen-ciphertext attacks (CCA), a standard attack model for cryptanalysis where the attacker has oracle access to the encryption function and an arbitrary collection of old ciphertext-plaintext pairs from the SE Enclave. This model is more powerful than that of chosen plaintext attacks (CPA). We assume that the attacker has bounded computational power and storage space, and does not have access to the encryption key a priori. If the encryption scheme used by the SE Enclave has indistinguishability under CCA (termed CCA Security), then an attacker has close to a random-guess success rate in learning the plaintext value of a new ciphertext released by the SE Enclave, even if the attacker has amassed an arbitrary collection of previous ciphertext-plaintext pairs. In the following sections, we detail the encryption scheme used by the SE Enclave and how it achieves CCA security.

The single-instruction CCA security described above immediately lifts to programs involving multiple instructions since CCA security in the single-message case implies CCA security in the multiple-messages scenario.

4.2 ISA Indirect Disclosure Safety Requirements

Another important requirement that the ISA specifies and the underlying design should enforce is that there cannot be any observable digital effects correlated to the plaintext value of the secrets. The most prominent indirect disclosure is based on the digital side channel of the program's timing behavior. Similarly, the control flow and memory access pattern of the program can produce information about private user data without direct disclosure. The ISA must restrict all three indirect disclosures.

4.2.1 Instruction Stream Side-Channels. It is necessary that knowledge of which instructions get executed does not yield any more information than what is known at compile time. Intuitively, such information can be derived only if there is a relationship between the control-flow structures of SE and the secret data of the user. However, insecure control-flow instructions do not see secret values by the definition of the ISA thus control-flow decisions cannot give any information about the secret. The only control-flow instruction that operates on secret data is the secure `CMOV` instruction. However, as a secure SE instruction, it always produces ciphertexts with the property described in §4.1.2. Unlike general control-flow instructions, like `jmp`, which influence which instructions are executed, `CMOV` always moves encrypted (and thus indistinguishable) data into the same destination register. Therefore, although `CMOV` makes a move decision based on secret data, it does not expose the secret. Consequently, observing the stream of executed instructions does not give the attacker a better-than-chance shot at guessing the secret. This property lifts to the program level from the instruction's property since all other disclosures from secure instructions to insecure instructions are removed by the direct disclosure restrictions from § 4.1.1.

4.2.2 Address Stream Side-Channels. Side channels in programs can also exist in their address streams where memory access patterns can expose information about private data indirectly. Similar to § 4.2.1 above, such indirect disclosure is possible only when there is a relationship between the user's private data and memory access instructions. However, the insecure memory instructions, such as `load` and `store`, do not have access to secret values. The only instruction that writes data to a location is the secure `CMOV` instruction, which we have argued in § 4.2.1 to be side-channel free.

Since all secure SE instructions do not disclose secrets by § 4.1.1 and the ciphertexts produced have the property described in §4.1.2, we conclude that all programs composed of SE instructions do not leak secret information through their address access streams.

4.2.3 Instruction Timing Side-Channels. A final possible side channel we consider is timing variance in secure instructions that can leak information about the secret data indirectly. First, we require all secure instructions not to have timing variance that correlates to secret data. By the direct disclosure argument in §4.1.1 we know

that insecure SE instructions do not have access to secret data thus any timing variance they might have in their execution (including nondeterministic variance) cannot be correlated to secret values. From these two properties of instructions, we can lift to program-level properties naturally. Specifically, a program composed of SE instructions executed on two different secret data will have exactly the same timing behavior if we fix the rest of the execution environment. All variations that might occur in this setting would be caused by insecure instructions possibly exhibiting nondeterministic behavior, which is always independent of the secret data. Thus the timing of programs does not reveal any secrets.

4.3 Formalization of ISA Properties

While designing an ISA is already a challenging task that must carefully tread the interface between the hardware's behavior and the software's expectations, stating security principles at the ISA level is an even harder task. Fortunately, the SE ISA comprises a few unique instructions with precise semantics and security requirements. This allows us to give a formal treatment to some of the informal descriptions presented in § 4.1 and § 4.2 regarding disclosure and side channels, respectively. To formally reason about the ISA level properties we first start by defining a minimal syntax recognized by the secure SE processor. We allow an arbitrary but finite set of registers observable by the attacker except for the `keyReg` which is private to the secure SE Enclave. The instruction `enc` r_1 is the encryption instruction that encrypts the value in the parameter register and places the result back into the same register. We abstract all binary operations, *e.g.*, addition, subtraction, and shifts, to the instruction `bop` $\mathsf{r}_1, \mathsf{r}_2$ where some operation is performed on the values of the parameter registers, and the result is placed into the first parameter. Finally, a `cmov` is written as an if-else statement with a single assignment instruction as the body. Note that, unlike standard imperative languages, we do not allow conditional `if` statements with arbitrary bodies and there is no `while` construct either. The sequential composition operator c; p is a syntax restriction, forcing it to be right-associative. Enforcing it syntactically simplifies the semantics and type system, yet has no significant effect since sequential composition is associative. Figure 3 presents this syntax of the SE language.

$$
\begin{array}{rcl}
\langle \text{Registers } e \rangle & ::= & r_1 \mid \cdots \mid r_n \mid \texttt{keyReg} \mid b \mid [b] \\
\langle \text{Commands } c \rangle & ::= & \texttt{enc } r_1 \mid \texttt{bop } r_1, r_2 \mid \texttt{skip} \\
 & & \texttt{if } r_1 : r_2 \leftarrow r_3 \texttt{ else } r_2 \leftarrow r_4 \\
\langle \text{Program } p, q \rangle & ::= & c \mid c; p
\end{array}
$$

Figure 3: Minimal syntax for the SE ISA.

The semantics of the language is given by the small-step semantics in Figure 4. The semantics is entirely standard in imperative languages with the exception of the encryption and decryption operations and the time function t. First, all values are defined on finite length bits, *i.e.*, $bits = \{0,1\}^j$ where $j \in \mathbb{N}$ is a non-deterministically picked natural number, which allows bit-level operators to be well defined in `bop`. Bits of all zeros can be interpreted as a boolean false and, conversely, bits of all ones can be interpreted as a boolean true.

Next, we use a simple and well-known encryption scheme which we present here for completeness. Details of this construction can be found in introductory textbooks such as [45, 67]. Let *Func* be the set of all functions of type $\{0,1\}^n \to \{0,1\}^n$ and $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a keyed pseudorandom function inducing a distribution on *Func*. As usual, a bijective pseudorandom function P is a pseudorandom permutation. We will write F_k and P_k defined as $F_k(\cdot) = F(k,\cdot)$ and $P_k(\cdot) = P(k,\cdot)$. A strong pseudorandom function F is a pseudorandom function such that any polynomially bounded adversary $\mathcal{A}$ has a negligible advantage in differentiating between F and the random function F' even when given the inverse function F^{-1}. A strong pseudorandom permutation P is a bijective strong pseudorandom function.

Now we can define the encryption scheme used by the SE enclave. Let $\mathcal{SE} = \langle encrypt(\cdot,\cdot), decrypt(\cdot,\cdot)\rangle$. The encryption key k is uniformly picked from keys of length $s+n$. The message space is $\mathcal{M} = \{0,1\}^n$, *i.e.*, messages of length n. The encryption scheme is constructed as follows for any $m \in \mathcal{M}$:

$$
\begin{array}{ll}
u \leftarrow uniform(s) & encrypt(k,m) = P_k(m||u) \\
decrypt(k,m) = P_k^{-1}(m)[0:n] & decrypt(k, encrypt(k,m)) = m
\end{array}
$$

for any strong keyed pseudo-permutation P of block length $s+n$, where $||$ is defined as string concatenation. The random value u is sampled from the uniform distribution of strings of length s.

This scheme, $\mathcal{SE}$, is known to be secure against chosen ciphertext attacks (CCA-secure) because an attacker has only negligible advantage in learning about m given the ciphertext $encrypt(k,m)$. We give a formal definition of CCA security [74, Definition A.1] and show that $\mathcal{SE}$ is CCA-secure [74, Theorem A.1] in the full version of this article.

To simplify the semantics of the ISA, we implicitly make the assumption that the polynomially-bounded attacker $\mathcal{A}$ has a stricter bound on the number of queries it can make to the encryption oracle. Intuitively, this assumption guarantees that the probability of the event that the same random value u is used for two different encryption oracle calls is effectively zero. Given this assumption, we take the distribution of the ciphertexts to be truly uniform.[1] This assumption only allows us to avoid the more complicated probabilistic semantics that would be required to properly model this negligible chance and otherwise has no effect on the semantics.

In the ISA semantics, we assume there is a truly uniform function that can generate s bits for every encryption query and that a strong pseudorandom permutation is used. We justify these assumptions in §5.2 by discharging verification responsibilities to the RTL-level verification, as summarized in Table **??**. This leaves us with two tasks at the ISA level: *i)* ensure that u is of some pre-fixed length s on every encryption, and *ii)* ensure that a fresh u from the random number generator is used for each encryption query. We formalize these two tasks in the semantics in the rest of this section.

The state of the system is represented by a set of registers R containing values of $b \in bits$ and the syntactically decorated $[b]$. Marking values representing ciphertexts with brackets allows us to consider all ciphertexts to be effectively equivalent once we define

[1] This effectively means the encryption scheme is a perfect encryption scheme. Strictly speaking, the pseudorandom permutation of the encryption scheme needs to be swapped for a random permutation to achieve this. But, any polynomial attacker only loses a negligible advantage due to this swap.

the equivalence class. The function $\sigma : R \to bits$ provides the mapping. Let Σ be the set of state maps σ and allow $\sigma_1, \cdots, \sigma_n$ to range over Σ. There are three syntactic categories: programs, commands, and registers. The small-step arrow $\to_r$ produces bits by reading registers. We combine programs and commands into one syntactic category for the semantics, since all commands are programs; this makes the semantics more readable. The small-step arrow $\to$ for this combined syntactic category is defined over configurations of $\langle p, \sigma\rangle$. We write $u \sim uniform(s)$ to say the value u is sampled from the uniform distribution of strings of length s. The function $t : \Sigma \to \mathbb{N}$ models the timing behavior of the system by allowing an arbitrary finite function to decide how long it takes for the instruction to complete execution. In a concrete design, the time taken can depend on a number of system and instruction level properties thus we allow the system to make this decision based on the entire system state σ.

The SE language semantics is a restricted fragment of standard imperative languages. For example, the conditional move commands (`CMOV-T` and `CMOV-F`) reflect this in that the bodies are single assignment instructions instead of the traditional recursive bodies. Additionally, the assignment is forced to the same location in both the true and false branch of the conditional. There is no use of exposed `store` or assignment operator in the language, but one could mimic it using the existing operators with the restriction that only ciphertext is written to locations. The occurrence of register symbols is treated as usual as a free variable (`REG`). The encryption instruction (`ENC`) encrypts the value found in the parameter register and places it back in the parameter. Similarly, the binary operations instruction (`BOP`) first decrypts the values, and computes on the plaintext values using the semantic operator $\oplus \in \{+, <<, >>, -, \cdots\}$, then finally re-encrypts the resulting value using the *encrypt* relation, and writes the ciphertext back into the first operand register. Finally, the sequence operator (`SEQ`) takes single steps transforming the head of the sequence until the base case of `skip` is reached.

Since the `SEQ` is the only inductive rule in the semantics, the program structure is that of a list instead of a tree as usual. Consequently, the small-step semantics never diverges in its execution. This property is essential for the soundness proof in Theorem 4.2. The security reasoning is done within the type system thus keeping the semantics standard.

The type system used to reason about the security of information flow in the SE language is presented in Figure **??**. The types are generated by the following grammar

$$\begin{array}{rcl} \langle \text{Security labels } \ell\rangle & ::= & \texttt{public} \mid \texttt{private} \\ \langle \text{Program Types } \tau\rangle & ::= & \langle \ell\rangle\ prog \mid \langle \ell\rangle \end{array}$$

Let $L = \{\ell_1, \cdots, \ell_n\}$ be the set of security labels and $\mathcal{L} = \langle L, \leq\rangle$ by the bounded security lattice generated by L. We assume $L = \{public, private\}$ and that *private* is the top of the lattice and *public* is the bottom of the lattice. Generally, any security lattice can be decomposed into a low and high partition so our assumption is without loss of generality. The typing environment Γ maps register locations to their types. We consider all register locations to be public except the `keyReg` which is marked as private, thus,

$$\Gamma(r) = \begin{cases} \text{private}, & r = keyReg \\ \text{public}, & \text{otherwise} \end{cases}$$

The rule `REG` simply states this in the inductive rules. All constant values from *bits* start out as public (rule `CONST`). Similarly, the `skip` instruction is always typed as public (rule `SKIP`). The instruction `seq` is typed as public if the two programs that are composed are already typed as public prog. The instructions `CMOV` and `BOP` are only required to demonstrate their operands are public and can be immediately typed as public prog.[2]

Next, we define an equivalency of states of the system and, specifically, low-equivalency following the convention in the literature [29, 60, 76].

Definition 4.1 (Low-equivalent)**.** First, on the finite length bits b and $[b]$, we define the equivalence class generated by the rules[3],

$$\text{EQUIV-BR}\ \frac{b, b' \in bits}{[b] \approx [b']} \qquad \text{EQUIV}\ \frac{b, b' \in bits \qquad b = b'}{b \approx b'}$$

[2]It is not always the case that functions with public inputs will output public outputs. These typing rules for CMOV and BOP are sound because the security features of the semantics ensures the output is encrypted with fresh salt.

[3]This equivalence class is justified by the fact that the ciphertexts have a uniform distribution. Thus to the attacker, any two ciphertext carry the same information and the attacker shouldn't have a reasonable preference between any two ciphertexts.

$$\text{CMOV-T}\ \frac{\begin{array}{c}\langle r_1,\sigma\rangle \to_r [c_1] \qquad \langle r_3,\sigma\rangle \to_r [c_3] \\ \langle keyReg,\sigma\rangle \to_r k \\ decrypt(c_1,k) = true \qquad decrypt(c_3,k) = m \\ u \sim uniform(s) \qquad encrypt(m||u,k) = [c_5]\end{array}}{\langle \text{if } r_1 : r_2 \leftarrow r_3 \text{ else } r_2 \leftarrow r_4,\ \sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip}, \sigma[[c_5]/r_2]\rangle}$$

$$\text{CMOV-F}\ \frac{\begin{array}{c}\langle r_1,\sigma\rangle \to_r [c_1] \qquad \langle r_4,\sigma\rangle \to_r [c_4] \\ \langle keyReg,\sigma\rangle \to_r k \\ decrypt(c_1,k) = false \qquad decrypt(c_4,k) = m \\ u \sim uniform(s) \qquad encrypt(m||u,k) = [c_5]\end{array}}{\langle \text{if } r_1 : r_2 \leftarrow r_3 \text{ else } r_2 \leftarrow r_4,\ \sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip}, \sigma[[c_5]/r_2]\rangle}$$

$$\text{REG}\ \frac{\sigma(r_1) = b}{\langle r_1,\sigma\rangle \to_r b} \qquad \text{ENC}\ \frac{\begin{array}{c}\langle r_1,\sigma\rangle \to_r n \qquad \langle keyReg,\sigma\rangle \to_r k \\ u \sim uniform(s) \qquad encrypt(n||u,k) = [c_1]\end{array}}{\langle \text{enc } r_1,\sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip}, \sigma[[c_1]/r_1]\rangle}$$

$$\text{BOP}\ \frac{\begin{array}{c}\langle r_1,\sigma\rangle \to_r [c_1] \qquad \langle r_2,\sigma\rangle \to_r [c_2] \\ \langle keyReg,\sigma\rangle \to_r k \qquad decrypt(c_1,k) = n \qquad decrypt(c_2,k) = m \\ u \sim uniform(s) \qquad encrypt((n \oplus m)||u,k) = [c_3]\end{array}}{\langle \text{bop } r_1\ r_2,\sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip}, \sigma[[c_3]/r_1]\rangle}$$

$$\text{SEQ}\ \frac{}{\langle \text{skip; q},\sigma\rangle \xrightarrow{t(\sigma)} \langle \text{q},\sigma\rangle} \qquad \frac{\langle \text{c},\sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip},\sigma'\rangle}{\langle \text{c; p},\sigma\rangle \xrightarrow{t(\sigma)} \langle \text{skip; p},\sigma'\rangle}$$

Figure 4: The small-step semantics of the SE Enclave.

Now we can define low-equivalence. In context Γ, states σ, σ' are low equivalent $\sigma \approx_l \sigma'$ if they are equivalent on all low locations,

$$\Gamma \vdash \sigma(r) \approx \sigma'(r) \text{ for all } r \text{ where } \Gamma(r) \leq l$$

We can now restrict the behavior of t using the definition above. We require that t decides the amount of time taken by the instruction only using `public` data,

$$\text{if } \Gamma \vdash \sigma \approx_l \sigma' \text{ then } t(\sigma) = t(\sigma') \tag{1}$$

Intuitively, this means the timing function t is influenced only by the location of level l or lower (*i.e.*, there is no timing dependency between the private data and the public data that could lead to timing side channels). *This corresponds to the instruction timing property in § 4.2.3.*

We just need one preliminary security lemma toward the main theorem now. For a well-typed program c, taking one step in the small-step semantics on two different states of the system that agree on publicly visible state locations will always produce states that continue to agree on publicly visible state locations. Moreover, the timing behavior of t will also be equivalent on the two final states. This means changing any private locations in the state will not have an observable change on the publicly observable locations of the output states or the timing behavior of the program. The attacker that can observe only public locations (thus not inside of SE) cannot tell the difference between two executions of a program where the private data might be different and thus cannot derive additional knowledge about the secret user data from the public data. This property corresponds to §4.1.1. Lemma 4.1 states this property formally now.

Lemma 4.1 (Single Step Security)**.** If

(1) $\Gamma \vdash c : \ell$
(2) $\Gamma \vdash \sigma_1 \approx_l \sigma_2$
(3) $\langle c, \sigma_1 \rangle \xrightarrow{t(\sigma_1)} \langle \mathrm{c}_1', \sigma_1' \rangle$
(4) $\langle c, \sigma_2 \rangle \xrightarrow{t(\sigma_2)} \langle \mathrm{c}_2', \sigma_2' \rangle$
(5) $dom(\Gamma) = dom(\sigma_1) = dom(\sigma_2)$

then we have $\Gamma \vdash \sigma_1' \approx_l \sigma_2'$ and $t(\sigma_1) = t(\sigma_2)$

Proof Sketch. By induction on structure of the derivation of $\langle c, \sigma_1 \rangle \xrightarrow{t(\sigma)} \langle \mathrm{c}_1', \sigma_1' \rangle$. The timing requirement is immediate from assumption (2) and Eq. 1. See the appendix in the full version of this article for the complete proof [74, Lemma B.1]. □

The security result can now be stated via the soundness of the type system. The soundness argument of Theorem 4.2 follows from Lemma 4.1 for the most part by generalizing the number of steps to an arbitrary number (*e.g.*, multiple steps until the program is equivalent to `skip`).

Theorem 4.2 (Soundness)**.** If

(1) $\Gamma \vdash c : \ell$
(2) $\Gamma \vdash \sigma_1 \approx_l \sigma_2$
(3) $\langle c, \sigma_1 \rangle \xrightarrow{n}_* \langle \mathsf{skip}, \sigma_1' \rangle$
(4) $\langle c, \sigma_2 \rangle \xrightarrow{m}_* \langle \mathsf{skip}, \sigma_2' \rangle$
(5) $dom(\Gamma) = dom(\sigma_1) = dom(\sigma_2)$

then we have $\Gamma \vdash \sigma_1' \approx_l \sigma_2'$ and $n = m$.

Proof Sketch. By induction on the number of steps. Both the base case and inductive case are consequences of Lemma 4.1. The timing property follows from the induction as well. See appendix [74, Theorem 4.2] for details. □

The type system and its soundness formalize the properties discussed in §4 except §4.2.1 (Instruction stream) and §4.2.2 (Address Stream). §4.2.1 and §4.2.2 refer to insecure instructions that can access all r where $\Gamma \vdash r$: public but cannot decrypt ciphertexts since `keyReg` is `private`. An insecure instruction can at most corrupt ciphertexts but when executed on two low-equivalent states must produce a pair of low-equivalent states as well. Any timing variation in these instructions is not correlated to private data since the behavior of t is restricted by low-equivalence. In the following section, we summarize the requirements the ISA places on the RTL-level to provide the security properties of Theorem 4.2.

4.4 Summary of Proof Obligations Discharged to RTL Verification

To achieve the above program-level properties, we expect the RTL implementation to satisfy some requirements about individual secure SE instructions. Note that we have no requirements for insecure instructions. We enumerate these requirements here and later show how the RTL-level verification formally guarantees these instruction-level properties. The properties are split into instruction-level properties which are properties that must be satisfied by each instruction, and a system-level property which is a more general requirement on the system.

5 RTL SECURITY PROPERTIES

In this section, we formally define the security properties in an RTL implementation of SE so as to support the instruction-level requirements from § 4. We employ standard *hardware information flow tracking (IFT)* to check RTL-level properties. Briefly, if secret variables (plaintext and crypto key) do not leak to outputs of the Enclave, then the attacker is not able to infer secret values based on any observation and thus security is guaranteed.

For the rest of this section, we will first discuss the security goal at the RTL level, clarify the connection between ISA-level assumptions and RTL-level properties, then provide a formal definition for the *hardware information flow* properties to be checked for the RTL design.

5.1 RTL-Level Security Goal

We assume that the attacker can observe and control any signal, register, and memory location outside the boundary of SE Enclave (including the Enclave's outputs), but they cannot observe and manipulate signals and states within SE Enclave. SE Enclave is connected with the rest of the system with a well-defined IO interface. Such an assumption keeps the footprint of trusted RTL design minimal.

Our RTL-level security goal is to prevent secret variables inside the Enclave from leaking to the outputs of the Enclave in two forms: functional leakage and timing leakage. Functional leakage happens if the attacker can directly infer secret information from the result of the SE Instruction. Timing leakage happens if the execution time

of some instruction depends on the secret and the attacker may infer secret information by measuring the execution time.

ISA-Level Requirement	RTL-Level Property
P1	Functional Correctness of Crypto and RNG
P2	No Timing Leakage at the SE Enclave Outputs
P3	No Functional Leakage at the SE Enclave Outputs

Table 2: Mapping between ISA-Level Requirement and RTL-Level Properties.

5.2 ISA-RTL Property Mapping

Table 2 provides the connection between requirements from the ISA-level and the properties to be checked at the RTL-level.

P1: The implementation is required to satisfy the requirements of the scheme $\mathcal{SE}$ defined in §4.3 without the restriction of the semantics on the number of calls the attacker makes. Thus, the implementation supports the strictly stronger attacker of the CCA-security game presented in the full version of this article [74, Definition A.1]. Two specific requirements are discharged on the cryptographic algorithms: the use of a strong pseudorandom permutation and availability of a truly random s-bit generator. For example, the RTL may implement AES-128 which is a block-cipher (thus $s = 64, n = 64$), and block-ciphers are an implementation of strong pseudorandom permutation [45, Ch. 3.6.4]. Other implementation options, such as RSA, come from the closely related family of trapdoor permutations and satify this requirement as well. The values u, which is defined in §4.3, is generated by a hardware-based TRNG as shown in Figure 2. As discussed in §3.3, we assume the existence of high-bandwidth and bias-free TRNGs and consider their design out of scope for this work. The quality of the random number generator can be checked using existing tools, *e.g.*, Dieharder [18]. Similarly, the functional correctness of the encryption and decryption units can be checked by existing techniques [48] and is not discussed in this paper.

P2: Any secret-dependent execution time will result in timing leakage, *i.e.*, secret dependent variation in the timing at which results are available at the SE Enclave outputs. Thus, we need to check there is no timing leakage at the SE Enclave outputs in the RTL.

P3: Any unencrypted secret at the data output would result in a functional leakage of the instruction result at the SE Enclave outputs. Thus, we need check there is no functional leakage at the SE Enclave outputs in the RTL. Further, without any information (functional or timing) about the encryption key (which is only in the SE Enclave), it is impossible to decrypt the ciphertext outside of the Enclave.

The notion of 'leakage' can be formalized using standard hardware IFT, which we will define for the SE Enclave and discuss in the following subsections.

5.3 SE Definition

We use the following finite state machine (FSM) to represent an SE Enclave: $SE = (I, O, S, S_0, N, F)$, where:

- I is a vector of input variables and the domain of I is $\mathbb{I}$.
- O is a vector of output variables. The domain of O is $\mathbb{O}$.
- S is a vector of state variables and the domain of S is $\mathbb{S}$.
- S_0 is the initial value for S.
- $N : (\mathbb{S} \times \mathbb{I}) \rightarrow \mathbb{S}$ is the next state function for S.
- $F : (\mathbb{S} \times \mathbb{I}) \rightarrow \mathbb{O}$ is the output function.

The secrets, *i.e.*, plaintext output p of decryption and the secret key k are also state variables in S. S also includes the random salt r used as input to the encryption unit.

To describe the execution of the SE Enclave, we introduce the notion of a finite-length execution trace: $\Pi = (\pi_I, \pi_O, \pi_S)$ represents an execution for n cycles where $\pi_I = (I_0, I_1, I_2, ...I_{n-1})$ is a trace of input values for each cycle, $\pi_O = (O_0, O_1, O_2, ...O_{n-1})$ is a trace of output values for each cycle, $\pi_S = (S_0, S_1, s_2, ...S_{n-1})$ is a trace of state variable values from each cycle. The elements in π_I, π_O, π_S follow the next state function N and the output function F. In the following parts of the paper, we may use $\pi_x = (x_0, x_1, x_2, ...x_{n-1})$ to denote the trace of a variable x which consists of the value of x at every cycle in the execution, and $\pi_Y = (Y_0, Y_1, Y_2, ...Y_{n-1})$ to denote the trace of a vector Y which consists of the value of Y at every cycle in the execution.

Next, we define information flow using execution traces.

5.4 Classic Information Flow Definition

The classic information flow definition is based on the well-known 'non-interference' property [21, 32]. The key idea of this property is: if a variable x never interferes with another variable y in the system, then replacing x with different values will never affect the value of y. Let s be a secret variable and o be an output variable. In our setting, if there is no information flow from s to o, then the value of o is independent of the value of s, thus it is impossible to infer s based on o.

s not influencing o means the value of o at every cycle is not changed when the value of s is replaced with an arbitrary value. Let Π be an execution trace of length n, Q be the vector state variables besides s, π_s denote the trace of s, π_Q denote the trace of Q, π_o denote the trace of o, F_o denote the output function for o, N_Q denote the next state function for Q. If in Π we replace π_s with a different trace π'_s (it differs from π_s in at least one cycle), then we can use π'_s to compute the new trace of Q, o, *i.e.*, π'_Q and π'_o as follows:

$$Q'_0 = Q_0, Q'_i = N_Q(I_{i-1}, Q'_{i-1}, s'_{i-1}), 1 \leq i < n$$

$$o'_i = F_o(I_i, Q'_i, s'_i), 0 \leq i < n$$

There exists information flow from s to o if and only if

$$\exists \Pi, \exists \pi'_s$$

such that after computing π'_Q, π'_o,

$$\pi_o \neq \pi'_o$$

(π_o, π'_o differ in at least one cycle)

However, we need to modify this classic information flow definition to analyze a design with encryption. In the threat model, we assume that the ciphertext after encryption will not leak any information about the plaintext. However, changing the plaintext will necessarily change the ciphertext regardless of the encryption scheme. Therefore, we would reach the conclusion that there is information flow from the plaintext to the ciphertext, but this would be a false alert. We describe our solution to this issue in the following subsection by using the notion of 'ciphertext declassification.'

5.5 Information Flow with Ciphertext Declassification

The idea of *declassification* [7, 68] allows information flow under a specific condition. This allows information flow to go through a certain variable such as the ciphertext output of the encryption unit under the condition that the encryption is finished. The similar idea is also applied in our ISA-level proof.

Let s be a secret variable, o be an output variable, and c be the ciphertext output of encryption. Let p be a predicate that represents the completion of encryption, *i.e.*, p is only true when encryption is completed. The intuitive way to realize declassification is to model the design such that when p is true, c is replaced by a free variable c_f. This cuts off the connection between s and c when p is true and blocks the information flow under this condition, but not otherwise.

Next, we will give a formal description of the above method. We construct a new FSM SE' from FSM SE as follows. (i) We add a free variable c_f to the input vector I. (ii) For the output function F and next state function N, replace all occurrences of c in their arguments using the following expression $p\ ?\ c_f : c$.

Denote the new input vector as I', the new output function as F', and the new next state function as N'. The only difference between FSM SE' and FSM SE is that in SE', c is replaced by c_f when p is true. Then, in the original FSM SE there exists information flow from s to o not going through c when p holds if and only if there exists information flow from s to o in SE'. We call the above technique 'ciphertext declassification information flow'.

5.6 Summary

Based on our threat model we want to check if there exists information flow from the secrets to the Enclave outputs not going through the ciphertext after encryption. We will demonstrate in § 6.2 how both functional and timing leakage can be captured using standard hardware IFT.

In the next section, we detail the SE implementations, including three secure implementations and four insecure implementations, then use hardware IFT to either detect leakage or prove security.

6 IMPLEMENTATION AND EVALUATION

To evaluate the effectiveness and performance of our RTL verification technique, we implemented a collection of SE Enclave designs with different microarchitectural optimizations or security flaws. The goal of the evaluation is to check if our verification technique is sufficient to support common microarchitectural design optimizations and catch security flaws when they are present. In this section, we first describe the designs we implemented as the verification targets and then explain our verification process on each of those design options, followed by experiment results.

6.1 SE RTL Enclave Design Details

In this subsection, we describe the details of the SE Enclave RTL designs and the flawed SE Enclave designs we used to validate our verification technique. Overall, we implemented seven different designs. Three of the designs are safe and valid designs including one default design, one area optimized design, and one design with our advanced decryption cache optimization. We also implemented four designs with different types of vulnerabilities in them to validate our verification technique.

Inst.	Description	Inst.	Description
Class: Shift		Class: Arithmetic	
SLL	Logic Left Shift	ADD	Add
SLA	Arith. Left Shift	SUB	Subtract
SRA	Arith. Right Shift	MULT	Multipliy
Class: Logical		MULTS	Signed Multiply
XOR	Logical XOr	Class: Comparison	
OR	Logical Or	LT	Less Than
AND	Logical And	LTS	Signed Less Than
Class: Encryption		Class: Conditional	
ENC	Encrypt	CMOV	Conditional Move

Table 3: Instructions Implemented in SE Prototype.

6.1.1 Implemented Instructions. All our prototype designs implemented 14 instructions in total as listed in Table 3. These 14 instructions fall in categories shown in Table 1. Instructions in class *Shift*, *Arithmetic*, *Logical*, and *Comparison* all take two 128-bit encrypted operands as inputs and generate one 128-bit encrypted output with a fresh salt value. ENC instruction takes one 64-bit plaintext input and generates a 128-bit output in its encrypted form. This instruction is to support adding a public plaintext value to an encrypted secret value. To do so, the developer first encrypts the public value with a fresh 64-bit salt and then carries out the operation with the corresponding instruction. CMOV is the only ternary instruction supported by SE Enclave. This instruction takes three operands as input: *condition*, *t_value*, and *f_value*. The output result is based on the value of the condition: *t_value* if the condition is true and *f_value* otherwise. For each of the designs described below, there are two outputs: Valid and Data. Valid is effectively a completion signal that indicates whether the value at Data is the valid result for the most recent instruction.

6.1.2 Default SE Enclave Design (Default). A simplified representation of the default design is shown in Figure 2. In this design, SE Enclave takes in three operands, an instruction, and a valid bit to signal the start of computation. On the output side, SE Enclave outputs a 128-bit always encrypted value with a valid bit signaling the end of the computation. We use a 10-round unrolled AES encryption and decryption unit. The key is stored in the key register within the SE Enclave. The ALU unit is a constant time unit that performs the computation on already decrypted plaintext values. For the purpose of evaluation, we used a Linear Shift Feedback Register(LSFR) with a random seed as our random number generator(RNG), which yields a 64-bit random number per cycle. Any random number generator that can generate 64 or more random bits per cycle can be a valid design candidate. The choice and implementation of an RNG are outside the scope of this paper.

The default design can be fully pipelined. The input ciphertext is fed into the decryption unit first. After decryption, the 64-bit plaintext value goes through the ALU for computation. The completed result is padded with a newly generated 64-bit salt from RNG before being sent to the encryption unit.

6.1.3 Optimized Architecture. We introduced two design variants of SE Enclave enabled by common optimizations: one optimized for area and the other optimized for performance.

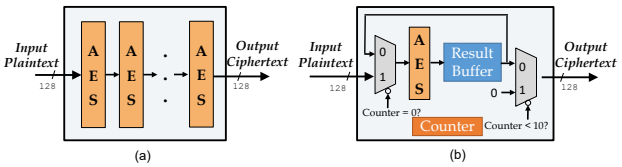

Figure 6: Unrolled AES (a) vs. Rolled AES (b)

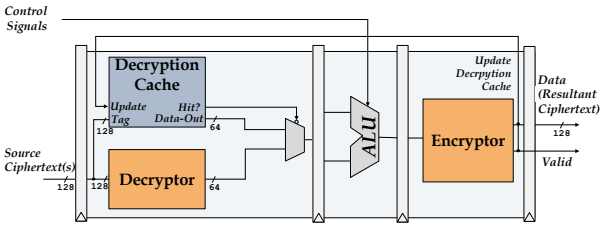

Figure 7: SE Enclave Design with Decryption Cache

Design with Rolled-Crypto Unit (Rolled AES): One optimization is to roll crypto units and thus make the SE Enclave an area-optimized architecture. The rolled AES uses a single shared register for each round and stores the intermediate results in the register after the completion of each round. The rolled architecture cannot be pipelined as it has only one register to hold intermediate AES encryption/decryption results after each round, thus the upstream data would need to be blocked until the completion of the full 10 rounds of AES encryption and decryption. Figure 6 shows the comparison between unrolled and rolled AES encryption.

Note that it is important for a rolled AES to prevent partially encrypted ciphertext from flowing outside of the encryption module because only fully encrypted ciphertext is considered secure.

Cache-enabled SE Design (Cache): Another optimization we implemented is to include a decryption cache as shown in Figure 7. One observation we have is that many applications demonstrate temporal locality — a recently computed result is more likely to be used as input operands in subsequent instructions. To exploit this type of locality, we cache the plaintext and the corresponding ciphertext for each result in the decryption cache as a First-in, First-out (FIFO) buffer. For each input operand, SE Enclave first looks up the input in the decryption cache and skips decryption if all operands are hit in the cache. We will show this type of design does not introduce side channels in the sections below.

6.1.4 Vulnerable Designs. To validate the effectiveness of our verification technique, we also implemented four microarchitectural vulnerabilities in various parts of the system that can leak information through side channels.

Exposed Partially Encrypted Ciphertext (Vulnerable Rolled AES): For the design with the rolled-crypto unit, a designer might be tempted to connect the output of the crypto unit directly to the register that holds the partial results of the AES encryption, and thus expose the partial encryption results to an attacker who can snoop that output. When not properly encrypted with a sufficient number of rounds, the partially encrypted ciphertext can be easily recovered through crypto-analysis [10]. This vulnerable design connects the output directly to the crypto result register to emulate a common mistake that might happen in the hardware design process. Thus, this causes a functional leakage from the SE Enclave.

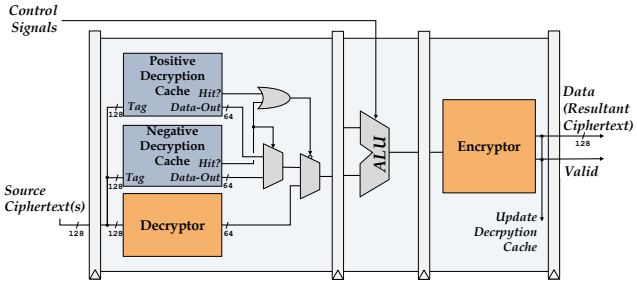

Figure 8: Flawed SE Enclave Design with Partitioned Decryption Cache

Value-dependent Timing Multiplier (Vulnerable Multiplier): The second vulnerability we introduced is a shift-and-add multiplier (sample code listed in the full version [74, Appendix D.1]), which causes a timing leakage. When one of the input operands is 0, the ALU in SE Enclave can immediately output 0 as the result of a multiplication operation; otherwise, the shift-and-add multiplication takes multiple cycles (operand dependent) to complete the computation. If the attacker measures the execution time of the `MULT` instruction for different encrypted operands, those finishing in fewest cycles indicate that one of the operands is likely to be 0, resulting in a timing leakage.

Value-dependent Cache Replacement(Vulnerable Cache): The third vulnerability we introduced is a value-dependent cache replacement policy, which leads to timing leakage. In this design, the cache is partitioned into two smaller caches, as shown in Figure 8. While still following FIFO, which cache the result would be placed in is dependent on the sign of the plaintext value. An attacker can conduct a known-plaintext attack [1, 44], similar to `Prime-and-Probe` [54], to first place a piece of data into one of the cache partitions. Then, the attacker replays the victim's instruction until the results would fill up one of the cache partitions. After that, the attacker probes the cache with the data they previously placed in the cache. By measuring the execution time of instructions to determine whether it is a hit or miss, the attacker can successfully recover the sign bit of victim data and even recover the full secret with enough trials through binary search as detailed in the full version of this article [74, Appendix C], which is similar to the approach used in other attacks, such as Blind ROP [14].

Value-dependent Timing RSA (Vulnerable RSA): Our last flawed design is a flawed RSA crypto engine (sample code listed in the full version of this article [74, Appendix D.2]). In RSA, to decrypt a piece of data, SE Enclave computes $m = c^d \ mod \ N$ where m is the decrypted message, c is the ciphertext, d is decryption key, and N is the modulus. To compute modular exponentiation effectively, the modular exponentiation module would compute bit by bit for the decryption key until it hits the most significant '1' in the key. This introduces a timing leakage because the decryption time depends on a secret (the private key) and the attacker can measure the execution time of instructions to gain information about the length of the key and potentially recover the whole key [80].

6.2 Evaluation Overview

6.2.1 Checking RTL Properties Using IFT. We check the RTL properties stated in Table 2 using standard IFT. The secret variables (the

source in IFT) are *i)* the plaintext after decryption and *ii)* the crypto key in the key register. The destination variables for the IFT are the two outputs of the SE Enclave, `Valid` and `Data`. Information flow to `Valid` indicates timing leakage as when `Valid` goes high, *i.e.*, the completion signal, depends on a secret. Information flow to `Data` indicates dependence of `Data` on the secret. This dependence may indicate functional leakage, *i.e.*, the secret may be inferred from the value of `Data` or timing leakage depending on the sequence of values at `Data`. For example, `Data` may stay at a value of `0` till it has a valid value, and switch to this valid value when it is ready. This distinction between functional leakage and timing leakage at `Data` is made by design analysis or using techniques from previous works [5, 61]. To distinguish this from the timing leakage observed at `Valid`, we refer to timing leakage at `Data` as functional-timing leakage.

6.2.2 Hardware Information Flow Tracking Tool. We can use any available IFT tool [6, 19, 39, 71, 75] to check for information flow. In this work, we use Cadence JasperGold Security Path Verification (SPV) tool [19, 39] for its availability in both industry and academia (through a university license). The tool will either prove that there is no information flow from the sources (the decrypted plaintext and key register) to the destination (the SE Enclave outputs) or find a hardware trace that demonstrates the information flow using symbolic model checking, *i.e.*, over all possible inputs.

6.2.3 Conditional Ciphertext Declassification. In our setting we need to check information flow with ciphertext declassification. SPV could potentially check this using *blackboxing* and the *not-through switch* (in previous SPV versions). SPV also allows specifying predicates on the source and destination under which information flow is allowed. However, in our setting we need to check information flow with *conditional ciphertext declassification* using predicates on intermediate signals, rather than with the source/destination (*e.g.*, only declassify the ciphertext when the encryption is finished). We accomplish this with a simple modification of the RTL design as described in § 5. Note that we need to treat unrolled encryption and rolled encryption differently. For unrolled encryption, because of its pipeline structure, we can directly replace the ciphertext output of the last crypto unit in the pipeline with a free input c_f because the output of the last crypto unit is always completely encrypted which means the completion predicate p is always true. For rolled encryption, since there is only one crypto unit, we only replace its output with a free input when the counter indicates that it is the last round. Thus, for a 10-round encryption, the completion predicate p is $Counter == 10$.

6.2.4 Cache Initialization. When evaluating SE variants with caching, we initialize the cache to be in an arbitrary state, *i.e.*, any cache line can be valid or invalid. Since JasperGold conducts symbolic model checking, all possible initial states will be explored, thus ensuring full coverage. This helps catch vulnerabilities in a much shorter time because some vulnerabilities only leak information when the cache is full. If we initialize the cache to be empty, the formal tool needs to conduct an extremely long symbolic execution, which significantly increases the verification time. If we initialize the cache to be full, although we may catch the vulnerability leaking information with the full cache faster, the tool cannot cover the entire search space because there is no instruction to flush the cache in the SE Enclave. However, we will fail to capture vulnerabilities that only leak information when the cache is not full. Therefore, initializing the cache to be in an arbitrary state can avoid the above two drawbacks. As the experiments demonstrate, the formal tool can prove security with full coverage, and also capture vulnerabilities efficiently.

In the following subsection, we will evaluate implementations of the SE Enclave and its variants, along with several buggy implementations using the JasperGold SPV tool.

Setup	Processor: two Intel Xeon 5222 cores Memory: 256 GB Tool: Cadence JasperGold 2021				
SE Variant	**register bits**	**result**	**leakage**	**time**	**memory**
Default	6544	secure	-	0.1s	1.6GB
Rolled AES	1412	secure	-	0.1s	0.7GB
Cache	12784	secure	-	0.1s	1.6GB
Vulnerable Rolled AES	1412	insecure	functional (plaintext → `Data`, key → `Data`)	109.4s	2.5GB
Vulnerable Multiplier	6737	insecure	timing (plaintext → `Valid`)	63.3s	4.7GB
Vulnerable Cache	12752	insecure	timing (plaintext → `Valid`)	402.4s	14.7GB
Vulnerable RSA	4328	insecure	timing, functional-timing (key → `Valid`, key → `Data`)	0.1s	0.3GB

Table 4: Experimental Evaluation

6.3 Experimental Results

The experimental results are provided in Table 4. For every SE design variant, we show the number of register bits as an indicator of the size of the state space. In addition to the verification result, *i.e.*, *secure* or *insecure*, we also label every insecure design with the leakage type, along with the source and sink of the information flow captured. The `Vulnerable Rolled AES` implementation has functional leakage where both plaintext secrets and the encryption key are leaked to `Data`. The `Vulnerable Multiplier` and `Vulnerable Cache` have timing leakage where the plaintext secrets are leaked to `Valid`. For `Vulnerable RSA`, there exists information flow to both `Data` and `Valid`. The leakage through `Valid` is timing leakage, but there is also timing leakage through `Data` as it is also leaking information about execution time as follows. The RSA crypto engine can only be implemented as a non-pipeline structure (the number of rounds is variable and depends on the decryption key). Thus, similar to rolled AES encryption, its output needs to be blocked when the encryption is ongoing. This means that when the ciphertext is not ready, it will output some default value such as all `0` to the `Data`. The attacker can infer the decryption key by measuring the number of `0`s between two non-zero ciphertext outputs which is timing leakage. As discussed in § 6.2, we distinguish between the timing leakage to `Valid` and `Data` by referring to the latter as functional-timing leakage.

In a pipelined AES encryption in the default SE Enclave design, the crypto engine outputs a fully encrypted ciphertext every cycle, thus only `Valid` carries timing information and this attack does not work. This explains why there is no information flow to `Data` for `Vulnerable Multiplier` and `Vulnerable Cache`.

As shown in Table 4, it takes less time and memory to prove no information leakage for secure implementations than to detect information leakage for the vulnerable ones. In secure designs, since the path from secrets to both outputs is cut off after declassification, SPV only needs to do a simple structural path check to prove the security. In comparison, SPV needs to do a state space search in

order to detect information leakage in a vulnerable design. The time and memory usage are also affected by the number of register bits and the complexity of the design. Across all implementations, the maximum verification time is less than 7 minutes. The experimental results demonstrate that our evaluation scheme is able to prove security or catch leakage precisely and efficiently.

6.4 Summary

In this section, we evaluated different SE Enclave variants using information flow checking. In general, our information flow definition and evaluation methodology work on any low-trust architecture with encryption. This is because low-trust architectures limit trust to only a small hardware enclave which facilitates formal verification, and the conditional ciphertext declassification we implemented can correctly deal with the information flow going through encryption.

7 RELATED WORK

7.1 Secure Hardware Architectures

There have been numerous works in designing secure architectures as microarchitectural flaws (and lack of security awareness at the architecture level) continue to be exploited by software attacks.

Trusted Execution Environments (TEEs) [3, 4, 16, 20, 22, 23, 30, 51, 58, 64, 69, 72, 73] have been widely deployed by mainstream hardware vendors in their server-grade CPUs to provide execution integrity and data confidentiality. Although the root of trust is also in hardware, TEEs such as Intel SGX [22] and Keystone [51] have a significantly different threat model as they do not eliminate timing side channels. Sanctum [23], MI6 [16], and Ascend [30, 64] aim to eliminate timing side channels, but they have much larger trusted computing bases than SE, making them hard to be formally verified.

Multiple works have been developed to mitigate leakage of data through timing side channels during speculative execution. Hyperflow [29] enforces data security properties by static flow analysis at the hardware construction time. It relies on a trusted label manager to assign correct labels to data. OISA [82] presents a timing-channel free ISA extension that uses tags to distinguish *Public* and *Confidential* data. It verifies both ISA and microarchitectural implementation through an abstract machine. Speculative Taint Tracking (STT) [84] and Speculative Data-Oblivious Execution (SDO) [83] are consecutive works that use runtime speculative taint analysis to eliminate timing side channels for safe speculative execution by delaying transient execution on the backend. DOLMA [55] introduces the principle of transient non-observability and delays executions that are reliant on speculative memory micro-ops. Software-hardware contract [36] formalizes software security requirements and hardware capabilities to ensure confidentiality for benign programs.

In comparison, we present the ISA and RTL level verification of a low-trust architecture that cannot leak secrets for any (even malicious) program, which is absent in the aforementioned works.

7.2 Functional Leakage and Timing Leakage

In this work, we use the role of signals to differentiate between functional leakage and timing leakage, *i.e.*, using hardware information flow tracking at `Data` for functional leakage and at `Valid` for timing leakage. This idea was discussed in a previous work [43].

Aside from using the role of signals, previous works also tried to use the observation of a signal in consecutive cycles (sequence) to differentiate between functional and timing leakage [5, 61]. Such techniques require more complex hardware information flow tracking logic and can separate timing leakage and functional leakage without needing to know the role of signals. However, since our work is interested in capturing both types of leakage, we choose to use the simpler role-based classification.

7.3 Hardware Security Evaluation Schemes

7.3.1 Type-Based Hardware Security Evaluation. SecVerilog [85] and its variants [27, 28] introduce new hardware design languages that allow developers to attach different security levels to hardware variables while programming, and also to define the rules for information flowing between different security levels. To declassify, these languages provide 'downgrading' syntax in the language to permit specific information flows. Static type checking is conducted to formally verify that the security policy is followed by the design.

While it is computationally faster, verification based on static type checking is not as precise as verification based on symbolic checking as in our work [46]. Static type checking may easily produce false positives due to its conservative nature. Further, it is inconvenient and error-prone for users to use a new language along with security labels in developing hardware.

7.3.2 Other Evaluation Schemes Based on Information Flow. Depending on threat models, different security properties have been proposed and verified. These properties are based on the basic information flow property and add different conditions and constraints according to the threat model. Attempts have been made to detect timing leakage with taint propagation [31, 61] by verifying properties such as 'constant-time execution'. Works based on Unique Program Execution Checking [25, 26] are aimed at checking information leakage through out-of-order execution, where information flow caused by in-order execution is ruled out by constraints.

A key advantage of our work is that we are able to verify security properties at the instruction level for all programs while other works do not provide the same guarantees.

8 CONCLUSIONS

This work demonstrates how the security of all programs running on low-trust architectures can be ensured by clearly defining the security requirements of their ISA instructions, formally specifying the consequent proof obligations for RTL implementations, and then executing these proof obligations using RTL formal verification tools. Further, it shows how these proof obligations cover functional as well as timing side-channel leakage. Finally, the small footprint of the trusted part of the implementation enables completing the formal checks using existing state-of-the-art formal verification tools - something that is currently not possible for non-low-trust architectures. We demonstrate our approach using the SE architecture where the ISA and the program-level proof are handwritten and the RTL-level verification done using off-the-shelf formal tools. Our experiments using seven different design variants shows that our approach is effective in proving the security of correct implementations and detecting flaws in buggy implementations.

ACKNOWLEDGMENTS

We thank Professor Paul Grubbs from the University of Michigan for lending his expertise in cryptography.

REFERENCES

[1] 2023. Known-plaintext Attack. (2023). https://en.wikipedia.org/wiki/Known-plaintext_attack

[2] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Callan, Alenka Zajic, and Milos Prvulovic. 2018. One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 585–602. https://www.usenix.org/conference/usenixsecurity18/presentation/alam

[3] Tiago Alves. 2004. Trustzone: Integrated Hardware and Software Security https://www.techonline.com/tech-papers/trustzone-integrated-hardware-and-software-security/. *White paper* (2004). https://www.techonline.com/tech-papers/trustzone-integrated-hardware-and-software-security/

[4] 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*, Vol. 13. ACM New York, NY, USA.

[5] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 147–154.

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

[7] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 207–221.

[8] Jim Attridge. 2002. An Overview of Hardware Security Modules. *SANS Institute. Information Security Reading Room* (2002).

[9] Roberto Avanzi. 2017. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology* 2017, 1 (Mar. 2017), 4–44. https://doi.org/10.13154/tosc.v2017.i1.4-44

[10] Navid Ghaedi Bardeh and Sondre Rønjom. 2019. Practical Attacks on Reduced-Round AES. In *Progress in Cryptology – AFRICACRYPT 2019*, Johannes Buchmann, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer International Publishing, Cham, 297–310.

[11] Mathieu Baudet, David Lubicz, Julien Micolod, and André Tassiaux. 2011. On the Security of Oscillator-Based Random Number Generators. *J. Cryptol.* 24, 2 (apr 2011), 398–425. https://doi.org/10.1007/s00145-010-9089-3

[12] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. 2015. The SIMON and SPECK Lightweight Block Ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2747946

[13] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, and Todd Austin. 2022. Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 73–84. https://doi.org/10.1109/SEED55351.2022.00014

[14] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 227–242.

[15] Matt Blaze, Joan Feigenbaum, and Angelos D Keromytis. 1998. KeyNote: Trust Management for Public-Key Infrastructures. In *International Workshop on Security Protocols*. Springer, 59–63.

[16] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. Mi6: Secure Enclaves in a Speculative Out-Of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.

[17] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. 2007. Provably Secure Authenticated Group Diffie-Hellman Key Exchange. *ACM Transactions on Information and System Security (TISSEC)* 10, 3 (2007), 10–es.

[18] Robert G Brown. 2018. Dieharder: A Random Number Test Suite. (2018). https://webhome.phy.duke.edu/~rgb/General/dieharder.php

[19] Cadence. 2022. JasperGold Security Path Verification. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html

[20] David Champagne and Ruby B Lee. 2010. Scalable Architectural Support for Trusted Software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[21] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[22] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *Cryptology ePrint Archive* (2016).

[23] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[24] M. Delgado-Restituto, A. Rodriguez-Vasquez, S. Espejo, and J.L. Huertas. 1992. A Chaotic Switched-Capacitor Circuit for 1/f Noise Generation. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 39, 4 (1992), 325–328. https://doi.org/10.1109/81.129465

[25] 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.

[26] 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.

[27] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. 2017. Secure Information Flow Verification with Mutable Dependent Types. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[28] 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*. 555–568.

[29] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. [n. d.]. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto Canada, 2018-10-15). ACM, 1583–1600. https://doi.org/10.1145/3243734.3243743

[30] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*. 3–8.

[31] Klaus v Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium (USENIX Security 19)*. 1411–1428.

[32] Joseph A Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11.

[33] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic Encryption amp; How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing* (San Francisco, California, USA) *(STOC '82)*. Association for Computing Machinery, New York, NY, USA, 365–377. https://doi.org/10.1145/800070.802212

[34] Louis Goubin and Jacques Patarin. 1999. DES and Differential Power Analysis the "Duplication" Method. In *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*. Springer, 158–172.

[35] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-channel Analysis of Cryptographic Software via Early-terminating Multiplications. In *International Conference on Information Security and Cryptology*. 176–192.

[36] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[37] Patrick Haddad, Yannick Teglia, Florent Bernard, and Viktor Fischer. 2014. On the Assumption of Mutual Independence of Jitter Realizations in P-TRNG Stochastic Models. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.052

[38] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. 2006. Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core. In *9th EUROMICRO conference on digital system design (DSD'06)*. IEEE, 577–583.

[39] Ziyad Hanna. 2013. Jasper Case Study on Formally Verifying Secure On-Chip Datapaths. https://www.deepchip.com/items/0524-03.html. https://www.deepchip.com/items/0524-03.html

[40] Ben Harris. 2006. *RSA Key Exchange for the Secure Shell (SSH) Transport Layer Protocol*. Technical Report.

[41] Simon Heron. 2009. Advanced Encryption Standard (AES). *Network Security* 2009, 12 (2009), 8–12.

[42] Russell Housley, Warwick Ford, William Polk, and David Solo. 1999. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile https://www.rfc-editor.org/rfc/rfc5280.html*. Technical Report.

[43] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware information flow tracking. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–39.

[44] Tony Sale James Wyllie. 1944. A Cryptographic Dictionary. *NR 4559, Historic Cryptographic Collection, Pre-World War I Through World War II, Record Group 457*

(1944). https://www.codesandciphers.org.uk/documents/cryptdict/cryptix.htm
[45] Jonathan Katz and Yehuda Lindell. 2020. *Introduction to modern cryptography*. CRC press.
[46] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S Foster. 2010. Mixing type checking and symbolic execution. *ACM Sigplan Notices* 45, 6 (2010), 436–447.
[47] Kyungduk Kim, Stefan Bittner, Yongquan Zeng, Stefano Guazzotti, Ortwin Hess, Qi Jie Wang, and Hui Cao. 2021. Massively Parallel Ultrafast Random Bit Generation with a Chip-scale Laser. *Science* 371, 6532 (2021), 948–952.
[48] Joseph R Kiniry, Daniel M Zimmerman, Robert Dockins, and Rishiyur Nikhil. 2018. A Formally Verified Cryptographic Extension to a RISC-V Processor. *Computer Architecture Research with RISC-V–CARRV 2018* (2018).
[49] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 388–397.
[50] Ting-Kuei Kuan, Yu-Hsuan Chiang, and Shen-Iuan Liu. 2014. A 0.43pJ/bit True Random Number Generator. In *2014 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 33–36. https://doi.org/10.1109/ASSCC.2014.7008853
[51] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[52] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel.. In *USENIX Security Symposium*. 717–732.
[53] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency Throttling Side-Channel Attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1977–1991. https://doi.org/10.1145/3548606.3560682
[54] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. https://doi.org/10.1109/SP.2015.43
[55] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1397–1414. https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin
[56] Stefan Mangard. 2003. A Simple Power-analysis (SPA) Attack on Implementations of the AES Key Expansion. In *Information Security and Cryptology—ICISC 2002: 5th International Conference Seoul, Korea, November 28–29, 2002 Revised Papers 5*. Springer, 343–358.
[57] Ueli Maurer. 1996. Modelling a Public-Key Infrastructure. In *European Symposium on Research in Computer Security*. Springer, 325–350.
[58] 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. *Hasp@ isca* 10, 1 (2013).
[59] Nissa Mehibel and M'hamed Hamadouche. 2017. A New Approach of Elliptic Curve Diffie-Hellman Key Exchange. In *2017 5th International Conference on Electrical Engineering-Boumerdes (ICEE-B)*. IEEE, 1–6.
[60] Andrew Myers. 2011. Proving noninterference for a while-language using small-step operational semantics. (2011).
[61] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging Gate-level Properties to Identify Hardware Timing Channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 9 (2014), 1288–1301.
[62] C.S. Petrie and J.A. Connelly. 2000. A Noise-based IC Random Number Generator for Applications in Cryptography. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47, 5 (2000), 615–621. https://doi.org/10.1109/81.847868
[63] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. 2020. Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. Cryptology ePrint Archive, Paper 2020/549. https://eprint.iacr.org/2020/549 https://eprint.iacr.org/2020/549.
[64] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2017. Design and Implementation of the Ascend Secure Processor. *IEEE Transactions on Dependable and Secure Computing* 16, 2 (2017), 204–216.
[65] Eric Rescorla. 1999. *Diffie-hellman Key Agreement Method*. Technical Report.
[66] Ángel Benito Rodríguez Vázquez, Manuel Delgado Restituto, Servando Carlos Espejo Meana, and José Luis Huertas Díaz. 1991. A Switched-Capacitor Broadband Noise Generator for CMOS VLSI. *Electronics Letters, 27 (21), 1913-1915.* (1991).
[67] Mike Rosulek. [n. d.]. The Joy of Cryptography. https://joyofcryptography.com https://joyofcryptography.com.
[68] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548.
[69] Matthias Schunter. 2016. Intel Software Guard Extensions: Introduction and Open Research Challenges. In *Proceedings of the 2016 ACM Workshop on Software Protection*. 1–1.
[70] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka Zajic, and Milos Prvulovic. 2020. A New Side-Channel Vulnerability on Modern Computers by Exploiting Electromagnetic Emanations from the Power Management Unit. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 123–138. https://doi.org/10.1109/HPCA47549.2020.00020
[71] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. In *31st USENIX Security Symposium (USENIX Security 22)*. 2549–2566.
[72] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 357–368.
[73] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. 2005. Design and Implementation of the AEGIS Single-Chip Secure Processor using Physical Random Functions. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 25–36.
[74] Qinhan Tan, Yonathan Fisseha, Shibo Chen, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. 2023. Security Verification of Low-Trust Architectures. *Long Version of CCS 2023 paper* (2023). http://arxiv.org/abs/2309.00181
[75] Mohit Tiwari, Hassan MG 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*. 109–120.
[76] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.
[77] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 679–697. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen
[78] Wei Wei and Hong Guo. 2009. Bias-free true random-number generator. *Opt. Lett.* 34, 12 (Jun 2009), 1876–1878. https://doi.org/10.1364/OL.34.001876
[79] Joel Weise. 2001. Public Key Infrastructure Overview. *Sun BluePrints OnLine, August* (2001), 1–27.
[80] Wing H Wong. 2005. Timing Attacks on RSA: Revealing Your Secrets through the Fourth Dimension. *XRDS: Crossroads, The ACM Magazine for Students* 11, 3 (2005), 5–5.
[81] S. Yasuda, H. Satake, T. Tanamoto, R. Ohba, K. Uchida, and S. Fujita. 2004. Physical Random Number Generator Based on MOS Structure after Soft Breakdown. *IEEE Journal of Solid-State Circuits* 39, 8 (2004), 1375–1377. https://doi.org/10.1109/JSSC.2004.831480
[82] Jiyong Yu, Lucas Hsiung, Mohamad El'Hajj, and Christopher W. Fletcher. [n. d.]. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *Proceedings 2019 Network and Distributed System Security Symposium* (San Diego, CA, 2019). Internet Society. https://doi.org/10.14722/ndss.2019.23061
[83] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 707–720. https://doi.org/10.1109/ISCA45697.2020.00064
[84] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274
[85] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A Hardware Design Language for Timing-sensitive Information-flow Security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.
[86] İhsan Çiçek and Günhan Dündar. 2013. A Chaos-based Integrated Jitter Booster Circuit for True Random Number Generators. In *2013 European Conference on Circuit Theory and Design (ECCTD)*. 1–4. https://doi.org/10.1109/ECCTD.2013.6662257