Hardware Security Leak Detection by Symbolic Simulation

Neta Bar Kama
Core and Client Development Group
Intel Corporation
Haifa, Israel
neta.bar.kama@intel.com

Roope Kaivola
Core and Client Development Group
Intel Corporation
Portland, OR, USA
roope.k.kaivola@intel.com

***Abstract*—Aiming to expose security risks in hardware designs, we describe a novel usage of symbolic simulation that led to discoveries of previously unknown potential local data leakages on an Intel Core processor design. Symbolic simulation is an established formal verification method, the main vehicle for verification of arithmetic data-paths in Intel Core processor designs for twenty years. It extends traditional simulation by allowing symbolic variables in the stimulus, covering the circuit behavior for all possible values simultaneously. A special trait of symbolic simulation is that every variable has a name. In the security context, named values allow us to know the exact origin of data and identify data leakages by determining whether values are expected to be read by an operation or present a risk. Leveraging the existing formal verification infrastructure and observing an operation's data dependencies we could identify local leaks without the need to have a complete functional specification for the operation.**

***Index Terms*—Security, Data Leakage, Formal Verification, Symbolic Simulation**

I. Introduction

Comprehensive formal verification of execution engines has been standard practice in virtually all Intel® Core™ processor development projects in the last two decades, and extensive infrastructure has been built to support these efforts. The technical basis of this work is symbolic simulation, a technology extending usual digital circuit simulation with symbolic values, representing sets of concrete values in a single simulation.

In the aftermath of the Spectre and Meltdown vulnerabilities, security has become a greater focus area for validation. In this paper we discuss a novel approach leveraging the existing formal infrastructure for Intel Core processor Execution clusters (EXE) to analyze potential data leakages, security violations where privileged data could be made visible to non-privileged parties. The approach is based on the special feature of symbolic simulation that stimulus values have names that can be used to uniquely relate a value to a specific signal and time.

Below we first discuss the concept of symbolic simulation and its use in EXE formal verification, and the security challenges in EXE. Then, we will describe the principles of our solution analyzing potential data leakages using symbolic simulation, practical considerations in the implementation of the solution over a live Intel Core processor development project, and the results of our experiments. With a moderate engineering effort, we were able to extend the existing formal environment with extra checkers detecting potential data leakages. On the one hand, this allowed us to verify the absence of data leaks for large classes of micro-operations, and on the other to identify several previously undiscovered local data leakage issues, where micro-operations unintentionally wrote back data that had been left behind in the internal state of the cluster by a previous micro-operation.

The closest counterpart to our work in the scientific literature or commercial tools is taint analysis [1], [2], [3], [4]. Like our approach, taint analysis tracks the propagation of values from one signal to another. However, taint analysis works by attaching extra information, the 'taint', to simulation values to track their progress, and requires extra engineering either in the simulator or in post-simulation analysis. In our approach values are tracked using the symbolic variable names already present in the symbolic simulation for the verification, and we only needed to implement a thin analysis layer on top of the existing collateral. Second, taint analysis generally assumes a static classification of signals to 'secret' and 'non-secret' and analyzes possible paths leaking secret values to non-secret signals. This does not adequately reflect the common design pattern of pipelined designs, like the EXE cluster, where the same signals are used to carry both secret and non-secret data at different times, and the notion of a 'secret' is relative to a micro-operation. To our knowledge, our work is among the first published explorations of the application of symbolic simulation into security verification of hardware designs (cf. [2], [5]).

II. Symbolic Simulation in EXE Verifiction

A. Symbolic Circuit Simulation

Digital circuit simulation is a standard tool in the arsenal of every working circuit design and validation engineer. Symbolic simulation extends this technology with the ability to carry out

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_11

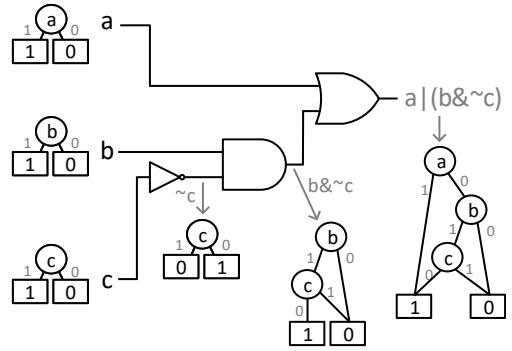

Fig. 1. Symbolic expressions in simulation

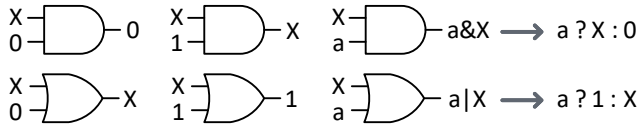

Fig. 2. Logic with the undefined value X

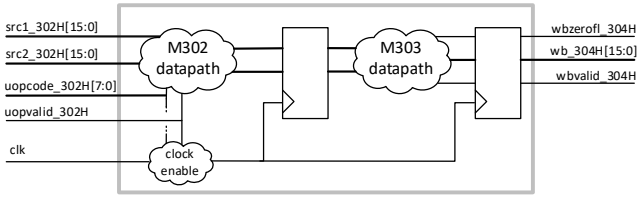

Fig. 3. Simplified ALU

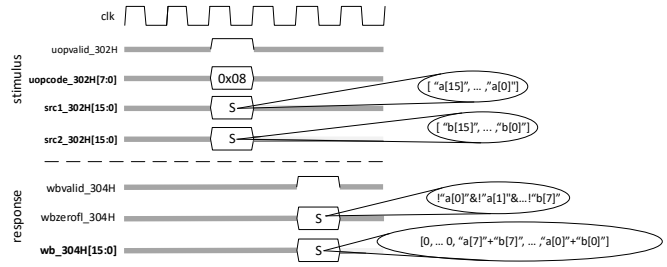

Fig. 4. Symbolic trace

a simulation using symbolic representations of sets of values in a single simulation trace [6], [7].

In a symbolic simulator the input stimulus may contain symbolic variables in addition to the traditional concrete values 0, 1, *X* or *Z*. These symbolic variables are effectively names of values, denoting sets of possible actual concrete values. In the simulation, these symbolic values propagate alongside the constant values, and in each logic gate, they may be combined with each other or one of the constants to result in either a logical expression on the symbolic variables, represented by an expression graph, or a constant. See Figure 1 for an example.

In a bit level symbolic simulator a single symbolic variable a corresponds to the set of Boolean values containing both 0 and 1. If stimulus to a symbolic simulation refers to the variables *a*, *b* and *c*, the internal signals might carry values like $a \wedge b$ or $a \vee (b \wedge \neg c)$. Usual logic rules apply: if the inputs to an AND-gate are *a* and 1, the output will be *a*, if the input to a NOT-gate is *b*, the output will be $\neg b$, and if the inputs to an AND-gate are *a* and *b*, the output is the logical expression $a \wedge b$. In symbolic simulation, a specific symbolic variable is associated with a specific signal and time in the stimulus. Associating a variable with a signal at a time does not fix the value, but instead gives a name that can be used to refer to the value.

In symbolic simulation, the constant value *X* is used to denote a universal undefined or unknown value, which propagates according to rules depicted in Figure 2. The value *X* denotes lack of information: we do not know whether the value is 0 or 1. The propagation rules reflect this intuition. Symbolic simulation uses *X*'s as an abstraction mechanism: unlike symbolic variables, *X*'s are an over-approximation of Boolean circuit behavior. Both symbolic variables and *X*'s allow us to verify a property over a single symbolic trace, and conclude that it is valid over every possible trace instantiating the *X*'s and the symbolic variables with 0's or 1's.

Figure 3 depicts a simplified pipelined ALU circuit with a 16-bit wide two-cycle data-path from sources to write-back. Figure 4 depicts a typical symbolic trace that might be used in the verification of this ALU, focusing on a single instance of an eight-bit wide bitwise OR micro-operation. The control signals are driven with concrete values corresponding to the operation, and the source data is driven with symbolic variables $a[15], \ldots, a[0]$ and $b[15], \ldots, b[0]$ in the one cycle in which the operation is issued. In all other cycles these signals are driven with the undefined value *X* (gray waveform). In the simulation, the values of the write-back data and zero flag two cycles later are then expressions on the symbolic variables associated with the source data.

A single symbolic simulation trace corresponds to a set of ordinary simulation traces, covering behaviors of the simulated circuit for all the possible instantiations of the symbolic variables with concrete values. The ability to cover all behaviors forms the basis of using symbolic simulation as a formal verification method. In this role symbolic simulation excels in verification of deep targeted properties of fixed length pipelines, typically of the transactional form *stimulus A at time t is followed by response B at time* $t+n$. It has a unique ability to carve out the circuit logic relevant to the progression of a pipeline while ignoring the rest of the circuit and other transactions in flight. As the approach is conceptually simple and concrete, it gives the human verifier a fine-grained visibility into the progress of the computation during a verification task, enabling precise analysis and mitigation of computational complexity bottlenecks. Because of these advantages, symbolic simulation can routinely handle circuits that are magnitudes above the capacity of more traditional formal property verification approaches, as well as circuits where the pipelines are too enmeshed to be amenable to equivalence-based verification methods.

B. Execution Cluster

Intel Core processor architecture has evolved gradually over the years. Typically, a new design project maintains functional backwards compatibility with earlier designs while providing improvements along different axes: new instructions and capabilities, improved performance or power, or design adjustments to meet side conditions set by a new manufacturing process. A design project routinely inherits components from earlier designs.

At high level, a single core consists of a set of major design components called *clusters*. The front-end cluster fetches and decodes architectural instructions, translates them to micro-operations and computes branch predictions. The out-of-order cluster receives streams of micro-operations from the front end, keeps track of dependencies between them, schedules ready-to-execute micro-operations for execution, takes care of branch misprediction and event recovery, retires completed instructions, and updates architectural state. The execution cluster carries out data computations for all micro-operations implemented by the design, performs memory address calculations, and determines and signals branch mispredictions. The memory cluster handles memory accesses, may contain first level caches and interfaces with a system-on-chip layer outside the core, including for example a graphics processing unit and a memory controller. The SystemVerilog source code of a cluster usually contains several hundred thousand lines of code. While not a physical entity like the above, microcode is also a major design component, the complexity of which is comparable to that of the clusters.

In this paper we focus on security validation of the execution cluster (EXE) on an Intel Core processor design. The EXE cluster consists of six main units: the integer execution unit (IEU) contains logic for plain integer and miscellaneous other operations, the single instruction multiple data (SIMD) integer unit (SIU) contains logic for packed integer operations, the floating-point unit (FPU) implements plain and packed floating-point operations such as DIV, MUL, ADD, etc., the address generation unit (AGU) performs address calculations and access checks for memory accesses, the jump execution unit (JEU) implements jump operations and determines and signals branch mispredictions, and the memory interface unit (MIU) receives load data from and passes store data to memory cluster, maintains store forwarding buffers, performs various datatype conversions, and takes care of data bypassing. In a typical contemporary Intel Core processor design, the EXE cluster implements over 5000 distinct micro-operations and supports multi-threading.

At an abstract level, the EXE cluster is a pipelined machine, receiving as input streams of micro-operations (micro-ops, uops) through a set of schedule ports. Each micro-operation receives its source data either through the cluster interface or through a bypass from a previous operation, and produces its result through a write-back port after an operation-dependent latency. The cluster has state components, which a micro-operation may read or update synchronously.

C. EXE Formal Verification

Formal verification of arithmetic data-paths has been a focus area at Intel ever since the Pentium® FDIV bug in 1994. The primary vehicle for this work is symbolic simulation, incorporated in Intel's in-house Forte verification toolset under the name of Symbolic Trajectory Evaluation (STE) [7]. Initially a research initiative during the Pentium Pro design cycle, Formal Verification has been carried out as a routine part of Intel processor development projects since Pentium 4 in 1999. All Intel Core processor EXE data-paths since 2005, as well as most Intel Atom® processor and Gen Graphics arithmetic engines have been formally verified using symbolic simulation [8], [9].

In concrete terms, EXE formal verification is carried out through a shared verification system called Cluster Verification Environment (CVE), a large software artifact that creates a standard, uniform methodology for writing specifications and carrying out verification tasks [8]. Underlying CVE is the Forte/reFLect toolset, consisting of the high performance simulator STE wrapped in a full-fledged functional programming language [7]. All verification takes place at the level of the full cluster, not the underlying individual units.

In verification of the EXE cluster, every micro-operation and every port on which the micro-operation can execute correspond to a separate symbolic simulation task. This simulation starts from a totally unconstrained initial state and focuses on one instance of the micro-operation under verification. The control signals that are relevant to the micro-operation are restricted according to the micro-operation, and the source data signals are driven with symbolic variables, as in the simplified example in Figure 4. Additionally, some internal and external control signals of the circuit are driven with symbolic variables and may be restricted using control invariants that are used to capture reachable state restrictions. Due to the unconstrained initial state of the simulation, such reachable state restrictions are not automatically accounted for in the verification and need to be manually formulated and separately verified. All other signals in the simulation are driven with the undefined value *X*. Altogether, in this setup the single instance of the micro-operation under verification in the single symbolic trace covers all possible invocations of the micro-operation in any legal trace of the circuit.

Effectively, in the verification setup for a single micro-operation the control signals are set to fix the data-path controls to match a single instance of that micro-operation, and symbolic variables on the data are used to exhaustively simulate the data-path instance. The simulation is then connected to an abstract functional reference model for the micro-operation through source and write-back mappings, and the output of the design and the reference model compared. These design-dependent mappings extract the intended source and result values for the micro-operation at the relevant times relative to the instance we are verifying.

For a large majority of micro-operations in the EXE cluster, the data-path can be exhaustively symbolically simulated in

one pass at the full cluster level. For certain complex operations like floating-point addition, careful case splits on the data space are needed to contain symbolic expression growth in the simulation, and for most complex operations like floating point divide or fused multiply add, a sequential decomposition strategy is applied.

III. EXE Security Verification

A. EXE and Data Security

Traditionally EXE validation has focused on the functional correctness of the micro-operations, including the validation of control logic required for non-interference from other operations simultaneously in flight. Since the Spectre and Meltdown vulnerabilities, security validation has become a greater focus area. In both exploits, a rogue process can theoretically gain access to privileged data by observing the side effects of speculative, although ultimately unsuccessful access to a memory location containing the secret. A key ingredient of these exploits is that secret data temporarily propagates and influences execution flows in the micro-architectural level, although the results of the computations on the secret data are appropriately squashed before they become architecturally visible. In the classic functional correctness sense this is not a problem, as the secret data is never directly exposed. However, in the exploits a rogue process tracks the ways in which the secret data has influenced the execution flows, especially through timing analysis, in an effort to statistically deduce the secret with a high probability. This means that we need to secure the propagation of secret data also at the micro-architectural level. As it is difficult to foresee all the ways in which the secrets' influences on execution could be exploited, the best strategy is to try to limit the propagation of secrets in the system as best as we can, and try to block any leakages at a local level as early as possible.

Looking at the EXE cluster from the security and data leakage perspective, the first thing to note is that in the larger context some micro-operations may be privileged, and some may not, some data may be secret, and some may not, but EXE has no awareness of that. All it sees are micro-operations and data. Privileged and less privileged operations are interleaved out-of-order in the same thread and between threads. The mixture of secret and non-secret makes it harder to formulate a property *Thou shalt not leak secrets*, as we don't have a good measure of what counts as a secret. However, each micro-operation has a well-defined notion of the data it is expected to process: which buses at which times relative to the operation carry its source and result data. Relative to an operation, we can then over-approximate all other data as secret. This leads to the following fundamental security property for EXE:

For every micro-operation executing in EXE, its result data should be exclusively a function of its source data.

By 'result data' we mean the main write-back data bus, flags, faults, and all auxiliary outputs together. This security property can be formalized more accurately as:

For every micro-operation u, there is a function $spec(u)$ such that for every trace T of the circuit and every point t of T, if uop u is issued at point t of T and we write src for the source data of u and wb for the write-back data of u relative to the point t of T, then $wb = spec(u)(src)$.

For many micro-operations, this security property follows automatically from functional correctness. If the specification for the operation is fully defined for all possible source values, and we have verified that the implementation fully agrees with the specification, there is simply no logical possibility for the result data not to be purely a function of the source data. However, many operations have partially undefined results, where some result components are unspecified either for all or some source values. For example, some floating-point micro-operations do not fully support all possible source values, reverting to microcode flows for rare or hard-to-implement cases, leaving the result data undefined. Similarly, certain helper operations that are used only in specific microcode flows in contexts where some parts of the result are never used may leave these result components undefined. Designs take advantage of the undefined spaces, as they allow an implementation to be optimized without a need to maintain identical behavior in the undefined space. These undefined spaces provide an opportunity for a micro-operation to write back values that are derived from some other data than its sources, including possibly secret data that has been or is being processed by other micro-operations.

The most common scenario of data leakage in undefined spaces is when secret data processed by an earlier micro-operation lingers in some internal flops of EXE and is passed to the write-back bus as a later micro-operation's undefined result. In a fully pipelined machine where all clocks toggle all the time, this scenario cannot happen, as secret data stays in any pipe-stage for exactly the one cycle when it is being processed before being overwritten by the next wave of values. However, such always-toggling designs are a thing of the past. Qualified clocks are ubiquitous, and their use increases and becomes more fine-grained by every design generation because of power considerations. In many data-paths the clocks toggle at most once for each operation. This means that any secret data processed by an operation remains in internal flops in every pipe-stage, until the next operation executing in the same data-path clears it. In this context the security property above can be viewed as setting a security perimeter around EXE. Secret data can linger on inside the cluster but cannot be exported through the write-back bus by any micro-operation.

The general concept of the analysis of data leakages through undefined behavior is directly relevant for the prevention of Meltdown-type vulnerabilities, although the areas primarily contributing to Meltdown are outside our focus area in EXE. An essential part of Meltdown is transient execution after a faulting load micro-operation from an out-of-bounds memory location containing secret data [10]. While the problematic load micro-operation produces a fault due to an access check violation, it may, under certain circumstances, nevertheless

have read the secret value from the memory location and passed the value on to a subsequent flow that exposes the secret. The specification for a load micro-operation is likely to be of the form *if the load does not generate a fault, the writeback data will be the value held by the memory location pointed to by the sources, otherwise the writeback data is a don't-care*. Note that the naive specification, without the faulting condition and the don't-care space, is very unlikely to hold for any real implementation, as a load can fault for a variety of reasons, many of which prevent the routing of the memory data to the writeback. This undefined space in the specification allows the secret to be exposed, or conversely, as pointed out by Canella et al: *"...merely replacing the data of a faulting instruction with a dummy value suffices to block Meltdown-type leakage in silicon..."* [10, p 252].

B. EXE Security Analysis with Symbolic Simulation

Considering the fundamental security property formulated above, an extremely useful feature of symbolic simulation is that every symbolic variable can be uniquely related to the signal and time it was associated with in the stimulus. Each 1 in stimulus looks exactly like any other 1, each 0 like any other 0, but every symbolic variable carries immediately in its name the notion of which signal and time it originated from. The uniqueness of names and the setup of EXE verification allows us to re-phrase the security property as:

For every micro-operation executing in EXE, the symbolic expressions for its result data should only refer to symbolic variables associated with its source data, and should not allow the undefined value X.

This property is relative to the symbolic simulation task for the micro-operation, as outlined in Section II-C. The symbolic re-formulation of the security property guarantees the original version since the single symbolic simulation for the micro-operation is an over-approximation of every possible invocation of the micro-operation in any trace. This means that we can simply read the function $spec(u)$ required by the original definition, mapping source data to the result, from the symbolic expressions for the result data.

Another way of viewing the matter is that the symbolic expressions on the write-back signals fully capture all dependencies of the write-back on any signals in their fan-in cone. The constant values in the simulation do not matter in this respect. Since the symbolic simulation for the micro-operation over-approximates every possible invocation of it in any trace, every constant value in the symbolic simulation is also present in all these invocations. Consequently, the propagation of such constants in the simulation to the write-back cannot disclose anything about the internal state of the circuit that would not be universally true. As a technical restriction, in our work all case splits and decompositions used to alleviate verification complexity are on data and not on control signals and will not turn any symbolic variables on control signals to constants.

Notice that the symbolic formulation of the security property is not a property about the value of the result data itself. Instead, it is a property about the symbolic expression used to represent the value of the result data in the simulation, and the symbolic names that occur in that expression. Because it talks about names, not values, it is not something that could be coded in methods that describe properties of signal values, such as SystemVerilog Assertions.

When we run a micro-operation that has a fully specified result data, we naturally verify that it writes exactly the data we expect it to and nothing else, as otherwise the verification would fail. However, when there is an undefined space in the output, the situation is trickier because we don't know what value to expect. The use of named variables allows us to verify that the result data is a function of the source data without the need to say what that function $spec(u)$ is, i.e. **without needing to specify the expected result value**. This is very efficient when we are looking at the undefined space, where typically there is no good definition of what the result should be.

C. Implementation

Next, we describe in detail how this idea was implemented. In high level, named variables allow us to:

A) Sample the output of a DUT to get a list of named variables that have propagated to it and occur in the symbolic expression it holds. In the example in Figure 4, bit $[0]$ of the write-back data carries the expression $a[0]+b[0]$, referring to the variables $[a[0],b[0]]$. We call this list the *dependency list* of the expression.

B) Identify suspicious names in the dependency list. The CVE infrastructure has a known naming convention, so the variable name allows us to distinguish the data that we would expect to propagate from suspicious data. In the example in Figure 4, the names $a[0]$ and $b[0]$ are expected, since they are the named variables driven to the sources of the operating uop.

The security analysis has two outcomes. First, we can detect security vulnerabilities where they exist. Second, the absence of detected vulnerabilities for the vast majority of micro-operations provides strong evidence that no secrets can be leaked to the interface of the cluster through those operations.

Data propagation in the circuit is often gated by specific operations that exclusively enable the data flow. If that enabling is too short, and there is no mechanism that clears the data after the operation, it can hang there. Stale data becomes a security risk when another operation can read this data. In early stages of verification environment development for a new project, the validation focuses on pure data-path verification in a sterile environment, and as a simplification, disables power gating and lets clocks toggle freely. At this stage all data flows uninterrupted, and we cannot guarantee there are no leakages coming from stale data on a power-gated bus. Security verification analysis becomes effective and meaningful only when we enable all power optimizations in the formal environment. At the time we started this security initiative, this pre-condition was met in almost all areas of the design we were working on.

Formal verification of arithmetic data-paths in the EXE cluster is fully covered in CVE using symbolic simulation. We have specifications for all existing micro-operations and the infrastructure to run a full regression to collect any information needed for the extra layer of security check. This provided a solid base for our analysis, and an efficient process that led to interesting results in a short time. The process can be divided into three stages.

1) *Identify operations that have an undefined result.*
   As an example, in the simplified ALU in Figure 3 the write-back bus is 16 bits wide, but a shorter operation like the eight-bit OR only uses bits [7:0] for the result. The upper bits [15:8] could be left undefined, which might provide an opportunity for data leakage. For any micro-operation, CVE provides two different mechanisms for undefined results:
   - Each uop in CVE has a defined data type signature, which specifies useful static information about the shape of the sources and result of the uop, such as data size, data type (integer, floating-point), signed/unsigned etc. The source or write-back data can be of NULL type, meaning it is not used by the uop. For NULL write-back, the checkers will not sample the write-back bus at all in a simulation.
   - A uop may have a defined write-back datatype, but its specification may explicitly encode a don't-care space. For example, the data output of a divide operation could be defined as a don't-care when the divisor is zero. In this case the checkers will sample the output in a simulation but will ignore the value for the functional correctness check. In the eight-bit OR example, we could sample the full 16 bit write-back bus, but not necessarily check the upper eight bits, leaving them explicitly undefined.

   For both methods the existing CVE data structures allowed us to easily identify the set of uops that produce undefined results, creating a clear goal for the main security analysis. The first step in enabling the security check was to switch from the first method to the second one for all uops, to make sure we always sample the write-back bus: identify the uops using the first method, convert the NULL data signatures to a meaningful type, and incorporate the explicit don't-care space into the functional specification.

2) *Sample results and detect unexpected variables.*
   This stage is the heart of the process, using the existing symbolic simulation capability in the two steps above: A) Sample the output and extract the list of variables in the symbolic expression, and B) Identify suspicious variable names in the list. The ingredients of this stage are:
   - Every variable in the dependency list has a name.
   - Expected variables are the named variables associated with the source signals in the aligned source pipe-stage of the current operating uop, as discussed above. As sampled by the operating uop, they are considered safe.
   - All *X* values on the outputs are flagged, since the unnamed undefined value *X* cannot tell where it came from and is therefore inherently suspicious.
   - By convention, a driven variable that is not part of expected source data for a uop uses a name that is a combination of the signal and the time at which it was driven, for example: "SignalName@24".

   Given the values in the write-back bus, we check for *X*'s and query the variable dependency list for suspicious names. In the eight-bit OR example of Figure 4, there are no *X* values, and the dependency list includes only 'good' names such as $a[7]$ or $b[0]$.
   This check is fully automated, as the classification of variable names to good vs suspicious ones can be done mechanically based on existing information about the intended uop source interfaces and variable naming conventions.

3) *Trace the suspicious variables.*
   The presence of the undefined value *X* or a suspicious name in the dependency list does not yet automatically mean that what we see is real data leakage. By methodology, symbolic simulation uses a maximally uninitialized start state for the simulation, with all signals having the value *X*, and uses stimulus that drives *X*'s on most inputs to the circuit, overapproximating the real legal behaviors of the circuit. We need to trace the suspicious variable or *X*, see how it propagated to the write-back, and understand whether the path to the write-back is possible in the real operating environment of the circuit. This stage is like the debug process of any simulation, tracing the origin of a value in the circuit. We use a schematic viewer that shows symbolic values and trace the ones that we find interesting. In some cases, to better analyze a behavior, we strengthen the simulation to drive a variable at an internal signal that used to hold an unnamed *X* that may propagate to the write-back.

Consider for example the simplified ALU of Figure 3 and assume that the circuit is augmented with power gating logic that turns off clocks for the high eight bits [15:8] of the data-path for operations that only operate on the low eight bits [7:0] of data. If we now simulate an eight-bit OR operation on the circuit as in Figure 5, we might observe *X* values in bits [15:8] of the write-back as in Figure 6, instead of the 'good' result of Figure 4. Tracing back the *X* values on the write-back, we would find an internal flop with the output *X* and a clock that does not toggle, as in Figure 7. In the circuit, this flop will hold any value the previous operation has left there, presenting a leakage risk. To check whether this data really propagates to the output, we want to track a concrete named variable. To do this, we drive unique named variables "Src1[15]@23" ... "Src1[8]@23" to the internal flop as in Figure 8, and observe these variables in the write-back, as in Figure 9. Once we understand the leakage mechanism, we can

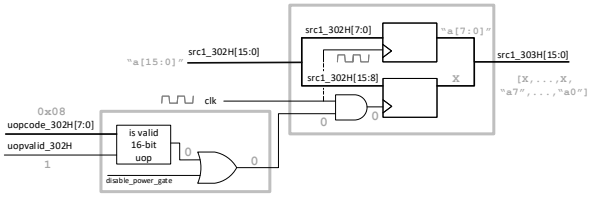

Fig. 5. Clock gating for eight-bit operations

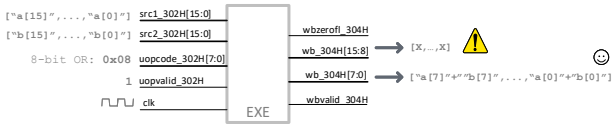

Fig. 6. Sampled *X* on the write-back bus

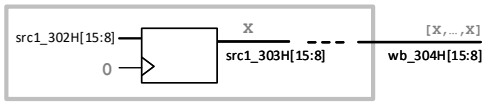

Fig. 7. Trace back the *X* to a gated clock

Fig. 8. Replace the *X* with a named variable

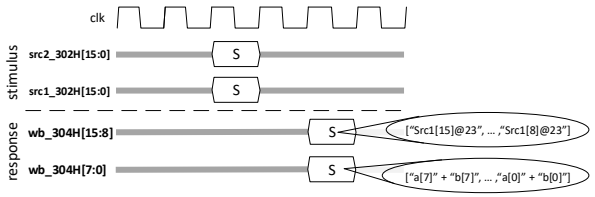

Fig. 9. Symbolic waveform with data leak

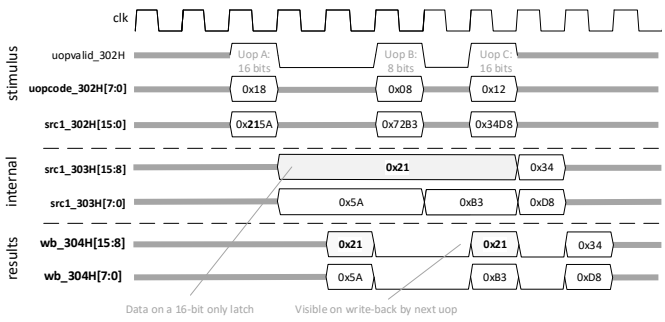

Fig. 10. Concrete waveform with data leak

then manually generate a concrete example exhibiting both an earlier uop leaving behind stale data, and a later uop that leaks the stale data to the write-back bus, as in Figure 10. In this example, the high eight data bits of a 16-bit uop A remain in the internal state until they are overwritten by the next 16-bit uop C, and are exposed by the 8-bit uop B in the meanwhile.

IV. Results

The flow of security verification was implemented as an automated extra check on top of the traditional data-path symbolic simulation. The process leveraged the existing capabilities of CVE that already supported all EXE uops. This gave us the ability to run a full regression and get first results quickly.

We chose to focus on the write-back data interface buses and concentrated on the about 2000 uops for which these buses are relevant, out of about 5000 legal uops for the cluster in total. Among these uops we first identified the ones that have fully or partially unspecified write-back data. Our analysis showed that 89.4% of the uops were completely specified, and 10.6% had unspecified write-back data. We then further analyzed the uops with unspecified write-back data by symbolic dependency analysis and found that 97.8% of uops were either completely specified or exhibited no unexpected data at write-back, whereas 2.2% of the uops had an undefined result space and failed the dependency analysis.

For the 97.8% of the uops that passed our analysis, we provided strong evidence that there is no risk of data leakage, as our analysis took place in the formal framework covering all possible behaviors. Note also that the dependency analysis allowed us to reduce the ratio of suspicious uops from 10.6% to 2.2%. As a restriction in scope, we did not look at data leakages in the bypass network, although the method would be equally applicable there.

The first real local EXE potential data leakage was discovered in less than a month. In a total effort of about two months of work, we discovered several different potential leakage mechanisms, all previously unknown. The failures were analyzed and grouped to RTL bugs with a common cause. Examples of potential leakage mechanisms include:

1) Uop A computed information intended to be written to the write-back data bus. It went through a latch that was toggling only while uop A was operating, for one cycle, and shut down right after uop A had completed. Therefore, the output of that latch was not cleared, and the data was stuck there on an internal bus. Analyzing uop B that was not expected to produce data (undefined write-back), we could see that uop A's data was propagating freely all the way to the write-back bus.
2) The data-path of a certain unit contained a MUX prior to the write-back bus with separate selects for specific uops and default logic shared by many uops. A particular uop C with undefined write-back executing in the unit read stale data left behind by any previous uop using the default logic.

3) Most uops that write only part of the write-back bus, for example 32 bits out of 128, have a clear definition of the unused bits, and we sample them along with the computed result of the lower part in regular data-path verification. In one exception, the upper part for a specific uop D was left unspecified. Tracing back the write-back, we reached an internal source bus shared by several operations, with a clock toggling just once per uop, causing the data to hang. Usually, the next uop would clear the bus. Uop D did not, leaking the upper bits of the source data left behind by the previous uop.

These bugs were all reproduced in normal simulation. They did not cause a functional failure: the results are never checked since they fall into the don't-care space of the specification. However, it was clear that the value written to the write-back is exactly the value left behind by a previous uop.

After the detection of these kinds of potential data leaks, there are several options for actions to fix them. The straightforward solution is to modify the currently undefined uop to have a defined value, e.g. write zeroes to the write-back data. This will be the easiest to verify because it will become again a strongly defined data-path verification task. It will also be the strongest solution, as it truly closes the leak. Another solution is to clear the stale data left by the earlier uop, for example by opening the gating clock for an extra cycle. Both options close the leak at the EXE boundary but require changing the design and could cost power or area.

If it is not possible to fix the design, another option is in the microcode level, making sure the undefined operation is not used in any way it could be exploited. Effectively here one establishes a security perimeter with a larger scope than EXE to see that the compromised data is contained before it becomes visible through a vulnerability at a higher level. This method is less optimal than the ones above, as the analysis scope is larger, outside the scope of existing formal tools, and relies more on finding parallels with known vulnerabilities, while new ways of exploiting information leaked out of the cluster may emerge. Also, micro-code implementation is dynamic, and it is possible that changes to the usage model that is safe today may make it unsafe tomorrow.

The potential local data leakages discovered by our analysis were addressed during the design project and as a result do not lead to a security violation at a user visible level in the final product.

V. Summary

Symbolic simulation's special trait — the usage of named variables — makes it a productive method to analyze data leakage risks. The scope of this work was huge for any formal analysis: a whole cluster, thousands of operations, and hundreds of thousands of flops in the circuit. Out of those, without having any prior knowledge where to look for the risks, we hit the relatively few instances that mattered in a short time. We found real issues, in a live project, issues that were not detected by any other method.

In this paper we described how we leveraged the existing environment of CVE that already supports the thousands of specifications in EXE cluster, holds information about data types and has a clear naming convention. This made the process efficient and demonstrated the importance of the complete verification environment covering EXE data-path. It is also important to clarify that the general concept we describe here is not dependent on it. Security verification by symbolic simulation can be implemented in various designs, where we do not have such infrastructure to rely on. Symbolic simulation is the key in analyzing data leakage risks of this kind, not the formal environment in itself.

In future design projects, with the increasing demand for security validation, we hope to explore where we can further develop this usage of symbolic simulation.

Acknowledgements

We would like to thank Arkady Neyshtadt for his security analysis, Gilad Holzstein, Robert Jones, Alex Levin, Yoav Moratt and Nir Shildan for discussions on security, Annette Upton for detailed feedback on the paper, and David Turner, Yaniv Dana and Alon Flaisher for the opportunity to carry out this work.

References

[1] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pp. 288–28815, 2019.

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, pp. 317–331, 2010.

[3] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–2, 2014.

[4] "Cadence JasperGold Security Path Verification (SPV) App," 2021.

[5] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 337–342, 2016.

[6] C. H. Seger and R. E. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods Syst. Des.*, vol. 6, no. 2, pp. 147–189, 1995.

[7] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, 2005.

[8] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *Computer Aided Verification* (A. Bouajjani and O. Maler, eds.), pp. 414–429, Springer Berlin Heidelberg, 2009.

[9] A. Gupta, M. V. A. KiranKumar, and R. Ghughal, "Formally verifying graphics FPU," in *FM 2014: Formal Methods* (C. Jones, P. Pihlajasaari, and J. Sun, eds.), pp. 673–687, Springer International Publishing, 2014.

[10] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 249–266, USENIX Association, Aug. 2019.