Toward a Methodology for Unified Verification of Hardware/Software Co-designs

Florian Lugou[1], Ludovic Apvrille[1], and Aurélien Francillon[2]

[1] Telecom ParisTech, Sophia Antipolis, France,
{firstname.lastname}@telecom-paristech.fr
[2] EURECOM, Sophia Antipolis, France,
aurelien.francillon@eurecom.fr

Abstract. Critical and private applications of smart and connected objects such as health-related objects are now common, thus raising the need to design these objects with strong security guarantees. Many recent works offer practical hardware-assisted security solutions that take advantage of a tight cooperation between hardware and software to provide system-level security guarantees. Formally and consistently proving the efficiency of these solutions raises challenges since software and hardware verifications approaches generally rely on different representations. The paper first sketches an ideal security verification solution naturally handling both hardware and software components. Next, it proposes an evaluation of formal verification methods that have already been proposed for mixed hardware/software systems, with regards to the ideal method. At last, the paper presents a conceptual approach to this ideal method relying on ProVerif, and applies this approach to a remote attestation system (SMART).

Keywords: hardware/software co-design, embedded system verification, security, ProVerif, tools

1 Introduction

Embedded systems are becoming more and more present in our daily lives. Many are now connected to the internet, even when used for vital functions. What used to be a concern for privacy has now turned into a requirement of strong security guarantees in critical systems. Only formal verification of these designs gives a mathematical guarantee of security.

It's not as unusual, now, to see medium-sized projects use custom hardware modification as it used to be. For instance some new projects – e.g.: the ESP8266 Wi-Fi chipset – don't use general-purpose CPU but prefer ASIPs like Processor Designer from Synopsys or Xtensa from Cadence. In particular, research topics have recently shown great interest in hardware-assisted security solutions ([3, 11, 21, 19]). In such designs, the overall security of the whole design relies on a tight cooperation between the customized hardware and the software running on it.

To illustrate the problem and guide our reflexion, we chose a hardware/software co-design that, we thought, was representative of many other hardware-assisted

security solutions which would greatly benefit from formal verification. This design is SMART, which stands for Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust and has been presented in [15]. This primitive tackles the problem of remote attestation by relying on a slightly customized microcontroller unit and a critical routine stored in ROM.

The process of remote attestation aims to detect devices that have been compromised, usually by computing a hash of the part of the memory to assess and sending back the result to a remote verifier. SMART provides the guarantee that this hash couldn't be calculated by the compromised device in any other way than correctly computing it.

In SMART, the memory layout is augmented by adding two read-only sections, as presented in Figure 1. The first one contains a procedure referred to as $\mathcal{RC}$ and the second one contains a key $\mathcal{K}$. $\mathcal{RC}$ can be called in order to compute a HMAC on a memory range $[a, b]$ passed as an argument. This HMAC is keyed with $\mathcal{K}$ to prevent a compromised software from computing the hash itself.

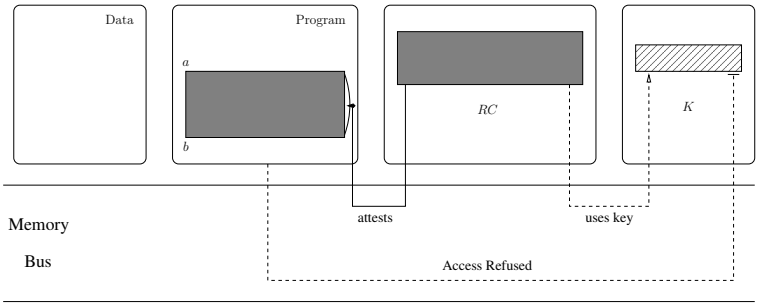

Fig. 1. SMART overview.

To be secure, the key $\mathcal{K}$ should be kept secret from the remaining – potentially compromised – software. This is guaranteed by adding a hardware protection which only allows access to $\mathcal{K}$ from $\mathcal{RC}$. Other hardware protection mechanisms were added but we will not consider them in this paper.

Here, the security of the system relies on the incapacity of a compromised device to forge a correct HMAC, which itself relies on the secrecy of the key. It is interesting to note that the secrecy of $\mathcal{K}$ is a simple property, but a naive modeling of the design would miss many possible attacks: what would happen if $\mathcal{RC}$ doesn't disable interrupts before loading the key in memory? What if control jumps in the middle of the routine? And what if the device is rebooted during the execution of $\mathcal{RC}$?

Applying formal verification to such designs thus requires us to take into account the non-standard hardware when analyzing software to prove system-level properties. To the best of our knowledge, no general methodology for unified verification has been found yet. Therefore we propose, as a first step, to survey

the different methods that have been applied up to now and also provide insights regarding potential new methods. This survey emphasizes the conceptual differences between two classes of verification methods: 1) methods that rely on the abstract concept of software to split the verification between the hardware part and the software part, and 2) methods that try to handle both at the same time by unifying the two concepts.

Our contributions are threefolds: a theoretical study of the problem of formal verification applied to the specific case of hardware/software co-designs, a survey of different methodologies that have been used up to now to verify such designs, and a tool that translates a subset of MSP430 assembly language into a ProVerif specification that ProVerif is able to handle.

In the second section we present the properties an ideal methodology for verification of hardware/software co-designs should have, then, in sections 3 and 4, we survey existing techniques. In section 5, we present our conceptual approach to the problem based on ProVerif, and finally, we discuss future work before concluding.

2 Expected Properties

In order to guide our reflexion and evaluate methods and tools, we list here the properties that one would expect from a formal environment when assessing the security of a hardware/software co-design.

2.1 Security-Aware Expressivity

Software has been steadily increasing in term of quantity and complexity in complete systems and is still undergoing considerable growth. Implementing critical functions in software may induce bugs or security flaws by increasing the attack surface and thus motivates the need to find solutions that guarantee properties, such as control flow integrity or code integrity, on any software. To target such global security properties new solutions often rely on specific hardware. The global security of the system thus depends on the security of the solution implemented as a tight mix of hardware and software. An efficient methodology dealing with this kind of designs should enable to express properties and give back results in a security-oriented meaningful way.

Natural Expression of Security Properties. First, verifiers would like to express what they want to prove as directly as possible. Translating the expected property into a combination of properties manageable by the solution (typically formulae in Conjonctive Normal Form), whose meanings are hard to grasp, is a source of errors. The verifier would thus be more interested by solutions that can naturally handle properties such as secrecy properties, taint propagation properties, etc..

Attacker Model. On the other side, expressing the capabilities of an attacker should be equally straight forward. It may be by using the "Dolev-Yao" model, or by tainting inputs that the attacker is able to control, for instance. The attacker model is normally coherent with properties the method is able to handle since the latter should be checked against the former, but a tool could also provide an automatic translation of abstract attacker models into low-level logic that the verification engine can handle.

Reconstruction of Traces. When the analysis tool determines that the required property may be violated, the designer must correct the erroneous part. The verifier should thus be able to rely on the feedback of the analysis framework to target the part of the design that would need to be re-designed. Since precisely and automatically determining the erroneous part of the design is currently impossible, a compromise often found is to provide the user with a trace summarizing the steps that lead to a state in which the property is violated. On the other side, returning the unsatisfiable core of a CNF formula would be of little interest for the designer.

2.2 Soundness of the Proof

Many hardware/software co-designs provide core features that are critical either for the proper functionning of the system (such as peripheral management), or for its security (e.g: access control, cryptographic primitives). These modules require strong safety and security guarantees that only formal verification is able to provide. Software analysis often has to deal with very large programs, which rules out complete verification. Here, we are concerned with smaller programs that hopefully enable us to mathematically prove that they are correct with respect to the features they were supposed to provide. Approximations are thus considered only as far as they don't affect the soundness of the proof.

In this sense, we are not directly interested in test-based co-verification of hardware/software co-designs as provided by commercial tools such as ZeBu [3], Seamless [4], or SoC Designer Plus [5].

2.3 Easy Adaptation to Hardware Modifications

When designing systems mixing hardware and software, one would need to see the effect of hardware modifications. Most verifications of software targeting embedded systems rely on a manual expression of the hardware model ([12, 23]). While finding a generic method that would deal with any hardware description may seem too optimistic, we believe that analysis of systems on chip would benefit from some modularity in terms of hardware models. We are thus interested in the extent to which each method can cope with hardware modification.

[3] http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/zebu-server-asic-emulator.aspx
[4] http://www.mentor.com/products/fv/seamless/
[5] http://www.carbondesignsystems.com/soc-designer-plus

3 Successive Verification of Hardware and Software

Traditional approach to hardware/software validation is to express a formal model of the hardware and use it during the verification of the software. The hardware may also be proved equivalent to the model, thus ensuring the overall security of the system.

For designs where hardware and software are tightly coupled, it may however be difficult to find an abstraction that would both enable the hardware to be verified, and require a manageable modification of a generic software analysis framework to integrate the specificities of the hardware. We discuss here how these two worlds could interface.

3.1 Expression of the Hardware Model

We target here designs where hardware and software must be checked together to ensure system-level properties. There are mainly two classes of such designs: either the hardware was customized in order to change the way the software was executed, or the hardware to verify doesn't affect the core processor but is a peripheral (such as MMU or a sensor), and the software part is handling the communication with this peripheral. In the first case, the software analysis tool would need to be modified to take into account the specificities of the hardware. In the second case, a common formal model could be found, and the hardware and the software could be checked separately against this model.

To prove that the hardware model – either when it is integrated into the software analysis framework, or when it is common to the software model – is a correct abstraction of the hardware, traditional verification of hardware designs could be applied. This verification is mostly done either by equivalence checking or by model checking. Many industrial and academic tools exist for this purpose: Vis, NuSMV, Incisive, Formality, etc..

3.2 Verification of Low-Level Software

Since we are here interested in both software and architectural vulnerabilities, we would like to take the compiler out of the trusted computing base. This is particularly true for security critical features - such as MMU management or cryptographic primitives - that are typically directly implemented as assembly code. Therefore, we are mainly interested in software verification tools that can take assembly code as input.

Using assembly code prevents us from working with higher-level concepts such as arrays, objects, functions, or types to guide the analysis. Losing such concepts means that we can't benefit from the semantic of coherent objects that the designer manually provided. For instance, it's simpler for the analysis to replace calls to a function by the formal expression that links the output of the function to its inputs and to prove that the function is indeed equivalent to this formal expression, than to analyse the function each time it is called in each context. However, we believe working with assembly code is more representative

of the attack scenarios we want to prevent (shellcodes, ROP, etc.) and of the software we want to verify.

In order to verify software at the assembly level, we ideally need a formal semantic of the instruction set. Such semantics are rare in practice, but progress has been done lately in this direction ([16, 12]). Once this formal model has been found, traditional software analysis methods may be applied: model checking ([8, 22]) and bounded model checking ([4]), symbolic execution ([18, 10]), etc..

3.3 Dealing with Hardware Customization

Since a designer may want to see the impact of a hardware modification as part of the design process, the amount of work needed to include this modification in the software verification framework and to prove the model to be a refinement of the modified hardware shouldn't be prohibitive.

One possibility would be to apply the CounterExample Guided Abstraction Refinement methodology ([9]): the tool would automatically compute a complex formal model of the hardware and use an abstraction of this model to analyze the software. If the targeted property were violated, a counterexample would be created and checked against the original hardware model. If it were spurious, the abstraction would be refined and the analysis would resume.

4 Unified Verification of the Whole Design

As disjoint verification of hardware and software naturally suffers from the considerable manual effort needed for finding a *good* abstraction that could both be proved to be a refinement of the hardware and be used as a base for verifying the software, some research work has been done to verify hardware/software co-designs as a whole ([20, 17]).

Similarly to successive verification of hardware and software, the methodology here also differs depending on how tightly the two are coupled.

4.1 Loosely Coupled Hardware and Software

For hardware/software co-designs where the processor is not modified and the verification effort should focus on some parts of the design that communicate with software through the use of a simple interface (memory mapped port, signals, etc.), the problem of hardware/software is no more vertical but horizontal, and it becomes possible to verify both at the same time, but still keep them distinct.

For instance, in [20], the authors propose a methodology for formally verifying a mixed hardware-software design implemented in SystemC. SystemC is a system-level language of which a subset may be synthetized. Thanks to their work, the SystemC specification can also be compiled to produce a formal model of the design (in the form of Labeled Kripke Structures). Parts of the design are recognized as hardware and the other parts are assumed to be software code.

4.2 Tightly Coupled Hardware and Software

Hardware based protections against software vulnerabilities often affect how the processor interprets machine code, by detecting policy violation as in [15], or by adding new instructions as in [3] for instance. In such designs, the processor itself is part of the verification target, and thus, simultaneous verification of disjoint hardware and software can't be done as in the previous section.

Including Software as Part of the Hardware Representation. In such case, we are trying to verify a customized processor implemented in hardware for a particular piece of software. However, we want to verify both as a whole, that is to say we don't want to create a formal model of the processor – probably because it would involve too much manual work. We can't prove the software with respect to a generic processor model since our processor is customized and we have no abstract model for our specific hardware. The abstract concept of *software* is thus unusable and program instructions must be considered for analysis in their true, concrete format: binary data. Concretely, this means filling the memory in the hardware representation with the program in binary format and verify this hardware as a whole.

Proving Properties on the Whole Design. Once software has been integrated into the hardware representation of the design, traditional hardware verification tools (such as model checker) could be used to prove the required property. However, some parts of the design are controlled by the attacker, typically some part of the memory corresponding to the procedure arguments could take any value. The value of these bits will affect how the program executes. For the analysis, this means that a huge number of states will have to be explored. Even for symbolic model checking it would be hard to automatically find good abstractions since the abstract states that would lead to a model being both provable and reasonably small would most likely relate software-level objects together. For instance let's say we are trying to verify a piece of software where a good abstraction – one that would make the property provable on the abstract model – would be *"the length of string s is smaller than the value of variable v"*. The objects s and v, however, have no more sense on hardware level, and automatically reconstructing them, by predicate abstraction for instance, would be difficult. Indeed, there is no hardware concept of what a string is, or what smaller means. For this reason such a verification scheme would probably be limited to very small programs.

5 Using ProVerif for Simple Symbolic Execution

For our case study, SMART, as we wanted to study the problem of formal verification of hardware/software co-designs from a generic point of view, we looked for a method which had some properties of the ideal method we described earlier and which could be adapted easily to other designs. That is to say we were searching a method that could:

– model a generic processor and instruction set,
– allow simple modeling of hardware customization,
– model an attacker and prove security-oriented properties,
– automatically produce a meaningful result, be it a clear answer if the property is proved to be true, or a trace if the property can be violated.

In the context of SMART, we could either model the whole device and the whole attestation protocol and try to prove that a compromised device can't forge a correct HMAC, or we could only model the routine $\mathcal{RC}$ and verify that $\mathcal{K}$ is not leaked. We will focus here on the secrecy of $\mathcal{K}$, but modelling the whole protocol should not add too much work.

5.1 Motivations for Using ProVerif

ProVerif [6] is a tool for analyzing protocols. It focuses on security protocols but its generic language (pi calculus) and simple reasoning with Horn clauses makes it a good candidate for a wide variety of applications [2].

Our requirements led us to search for a tool that would work with basic and generic logic and would target security properties. As ProVerif answered these needs, we chose it despite its original different field of application. As a result, this paper also presents a way to model the execution of a software and to prove properties on it using protocol analysis tools such as ProVerif.

An Interesting Attacker Model. The security of the SMART primitive relies on the secrecy of a key, and such a property can be natively represented in ProVerif. The tool also enables to query more complex properties such as authentication or observational equivalence.

These properties are checked against an attacker whose capabilities follow the "Dolev-Yao" model [13]. These kind of capabilities are also interesting in our particular design, where the remote verifier and the routine $\mathcal{RC}$ can be seen as participating in a protocol and the user controlled software on the device has full access to the abstract channel they are using.

A Simple Reasoning. ProVerif takes as input a description of a protocol in pi calculus language. This description is internally translated into Horn clauses that ProVerif uses for reasoning. Horn clauses are logical formulae of the form:

$$\bigwedge_i p_i \qquad \text{or} \qquad \bigwedge_i p_i \rightarrow q \tag{1}$$

where p_i, q are positive literals. The first formula corresponds to the case where there is no premise. This simple formulation makes it possible to model how each assembly instruction impacts the state of the system depending on the environment, and thus, allows for easy modeling of the effect of hardware modification on software execution.

Traces Reconstruction. Another feature of interest in ProVerif is its ability to reconstruct a trace when the queried property is violated. This trace is given as a succession of actions performed by the attacker that eventually lead to a violation of the property. The process of reconstructing this trace may fail (as explained in [1]) due to the approximations done when translating processes in pi calculus into Horn clauses. However, up to now, we managed to model our design to avoid this case.

Even though we haven't implemented it yet, we could automate the process of translating such a trace into a succession of software-related events that would make more sense in our context.

5.2 ProVerif Solving Algorithm

Our ambition was to prove properties on a relatively small piece of software running on a custom hardware. Since formally proving the property on a model of the software would mean exploring the entire state space, and sticking to a realistic model would limit the possibilities for abstraction, we assumed that our methodology would not scale well. Even though, we believe it may prove useful for small, central, security critical software, as it is the case for SMART.

We briefly present here how ProVerif is able to reason on specified protocols in order to explain how this is done for our model and compare the performance with more traditional techniques.

Horn Clauses and Predicates. Protocols that need to be verified by ProVerif are described as multiple processes that communicate between each other through private or public channels. The attacker can see anything that goes through public channels, intercept messages, create new ones, and send them on public channels.

The fact that the attacker knows about the message m is modeled as the predicate $attacker(m)$. The fact that a message m can be sent on channel ch is modeled as the predicate $mess(ch, m)$. As stated earlier, ProVerif works with Horn clauses so, for instance, the abilities of the attacker regarding channels are:

$$\begin{aligned} & mess(ch, m) \wedge attacker(ch) \rightarrow\ attacker(m) \\ \text{and} \quad & attacker(ch) \wedge attacker(m) \rightarrow\ mess(ch, m). \end{aligned} \tag{2}$$

Processes are also translated into Horn clauses. For instance a basic process and its translation are presented in Table 1. Note that both express the fact that if the attacker has knowledge of a he can acquire knowledge of $f(a)$. As it will be explained in the next section, this simple mechanism enables us to model an instruction-accurate version of a processor.

Clauses Unification. Once the protocol has been translated into Horn clauses, these clauses are combined to derive the total knowledge of the attacker. If the required property is violated during the process, a trace is computed based on the clauses that have been unified to lead to the violation.

Table 1. A ProVerif process and the corresponding Horn clauses.

ProVerif Process	Set of Horn Clauses

`process`
`  in (ch, a: bitstring);`
`  out (ch, f(a))`
	$mess(ch, a) \rightarrow mess(ch, f(a))$
	or $attacker(a) \rightarrow attacker(f(a))$
	if ch is public.

For our modeling, the way the clauses are unified will determine how the state space of the program is explored. Thus more information about the resolution process of ProVerif – as explained in [5] – is going to be exposed.

The idea behind clauses unification is to progressively expand the knowledge of the attacker. Let's say we have two clauses:

$$\begin{aligned} attacker(m) &\rightarrow attacker(f(m)) \\ \text{and} \quad attacker(f(m)) &\rightarrow attacker(g(m)). \end{aligned} \tag{3}$$

Unifying these two clauses is interesting since it will result in: $attacker(m) \rightarrow attacker(g(m))$, which means that if the attacker has knowledge of any message m, then $g(m)$ can also be known. By default, ProVerif considers that unifying two clauses is interesting when all the premises of the first clause are of the form $attacker(x)$ where x is a variable and when the premise of the second clause that can be unified with the conclusion of the first ($attacker(f(m))$ in our previous example) is not of the form $attacker(x)$. It means that it favors unifications that reduce the number of premises that are not of the form $attacker(x)$.

By unifying clauses like that, ProVerif eventually reaches a fixed point where no new clause can be generated. If the required property is the secrecy of a variable x, and eventually no clause of the form $attacker(x)$ has been derived, this is a proof that the attacker can't learn the value of x.

5.3 SMART Model

In the SMART design, the critical software part that we want to analyze is the routine $\mathcal{RC}$ that computes a HMAC with the key $\mathcal{K}$, and the attacker is a malicious software running on a corrupted device. It can access the whole memory and all the registers and may call $\mathcal{RC}$ as it likes. As the design relies on the secrecy of $\mathcal{K}$, we will only model the routine $\mathcal{RC}$, let the attacker define the state of the device before calling $\mathcal{RC}$, and check that $\mathcal{K}$ can't be leaked.

We show first how we automated the translation of MSP430 assembly code into a ProVerif model, and then, how hardware customization was integrated into the model. Finally, we demonstrate the solving process performed by ProVerif and relate it to a more classical software analysis method.

The Software Part. We model our software in an instruction-accurate way: we express the impact of each instruction on the state of the system. This semantic

enables us to consider attack scenarios and software designs that are realistic, especially on low level code: jumping in the middle of the routine, dynamic control flow graph with indirect jumps, etc.. Also note that since SMART was implemented for the MSP430 architecture, we modeled a subset of the MSP430 assembly language, composed only of basic instructions. For a more general approach, it would be better to use an intermediate representation such as REIL [14] or BAP intermediate language [7] and use the already existing front-ends to compile either assembly code or binary code to this intermediate representation.

Instruction at virtual address i is modeled as a process translated into a Horn clause of the form: $state(i, R, MEM) \rightarrow state(PC', R', MEM')$, where PC' is a program counter, R and R' are states of the registers, and MEM and MEM' states of the memory. PC', R', and MEM' are expressed as functions of R and MEM and model the effect of the instruction at address i on the state of the memory and registers – for instance $PC' = MEM[R[3]]$.

The $state(PC, R, MEM)$ predicate here would mean that a state of the system where the program counter is PC, the registers' values are R, and the memory is in state MEM is accessible. This predicate is obviously not defined in ProVerif and we must model it. We could do this by using a private channel: each message (PC, R, MEM) sent on the private channel $privch$ would mean that the state (PC, R, MEM) is accessible. The effect of an instruction at address i would thus be: $mess(privch, (i, R, MEM)) \rightarrow mess(privch, (PC', R', MEM'))$. However, private channels behave differently with respect to trace reconstruction. For instance, when trying to reconstruct a trace, ProVerif will only allow sending messages on a private channel if a process is ready to read the message on this channel. Therfore we chose to use an equivalent approach with public channels: $mess(ch, f(i, R, MEM)) \rightarrow mess(ch, f(PC', R', MEM'))$. Where f and its inverse un_f are private functions (with no explicit definitions) that guarantee that the fact $attacker(f(PC, R, MEM))$ – which means that the state (PC, R, MEM) is reachable – doesn't lead to $attacker(R)$ or $attacker(MEM)$, and reciprocally that the attacker can't create $f(PC, R, MEM)$ with any PC, R and MEM. Eventually the corresponding process in pi calculus is:

```
process
    in (ch, state: bitstring);
    let (PC: int, R: registers, MEM: memory) = un_f(state) in

    if PC=i then

    out (ch, f(PC', R', MEM'))
```

This process only models one instruction. To model the entire program, we created one process per instruction and replicated it – using ProVerif operator ! – so that the instruction could be invoked many times (in case of loops for instance). We wrote an open-source python script[6] that automates the process of translating MSP430 assembly code into a set of such processes.

[6] available at `https://gitlab.eurecom.fr/Aishuu/smashup`

The Hardware Part. The effect of hardware modification would here be included manually. However, it would be possible to limit the scope of possible hardware customizations (for instance by letting the designer define a policy for controlling the accesses to the memory). In such case, we could automate the translation of hardware specificities into a ProVerif model. Restricting access to part of the memory could easily be done by modifying the memory management functions. Enabling interrupts could also be modeled by including an interrupt vector in the state and by allowing the execution of an instruction only when no interrupt is pending.

Parallel with Symbolic Execution. The algorithmic efficiency of ProVerif resides in its ability to derive the complete knowledge of the attacker with as few clauses' unifications as possible. The policy used to choose which clauses to unify will guide the exploration of the program in our context. We'll show how this works on a basic example:

```
0       mov.w       #0x0000,    r4
    l0:
1       add         r3,         r4
2       sub         #1,         r3
3       jnz         l0
4       ...
```

As will be explained later, ProVerif has originally no representation for numbers. We will here ignore this fact and use them as intuition dictates. We will also only consider the first five registers and no memory to shorten the clauses, use $R2$ as the zero flag (instead of just one bit), and ignore overflows. For the sake of simplicity we will use $state(PC, R)$ as a shortcut for $mess(ch, f(PC, R))$ as was done before. Under these assumptions, the Horn clauses generated for the instructions would be:

$$\begin{aligned}
state(0,(R1,R2,R3,R4)) &\rightarrow state(1,(R1,R2,R3,0))\\
state(1,(R1,R2,R3,R4)) &\rightarrow state(2,(R1,R2,R3,R3+R4))\\
state(2,(R1,R2,1,R4)) &\rightarrow state(3,(R1,1,0,R4))\\
R3 \neq 1 \wedge state(2,(R1,R2,R3,R4)) &\rightarrow state(3,(R1,0,R3-1,R4))\\
state(3,(R1,0,R3,R4)) &\rightarrow state(1,(R1,0,R3,R4))\\
R2 \neq 0 \wedge state(3,(R1,R2,R3,R4)) &\rightarrow state(4,(R1,R2,R3,R4)).
\end{aligned} \tag{4}$$

If we allow execution of the routine only from the beginning this would add a clause:

$$\begin{aligned}
&attacker(R1) \wedge attacker(R2) \wedge attacker(R3)\\
&\wedge attacker(R4) \rightarrow state(0,(R1,R2,R3,R4)).
\end{aligned} \tag{5}$$

As mentioned earlier, ProVerif will only unify two clauses if the premises of the first one are all of the form $attacker(x)$. In our context, this means it will start the unification with the clause describing how the attacker could call the routine

(the last clause given above). Its conclusion is of the form $state(0, ...)$ so it could only be unified with a clause with a $state(0, ...)$ premise (the first one). Unifying these two clauses will result in:

$$attacker(R1) \wedge attacker(R2) \wedge attacker(R3) \rightarrow state(1, (R1, R2, R3, 0)). \quad (6)$$

Once again, this clause is the only one that could be used for unification so it will be unified with the clause corresponding to instruction 1:

$$attacker(R1) \wedge attacker(R2) \wedge attacker(R3) \rightarrow state(2, (R1, R2, R3, R3)). \quad (7)$$

Here, this clause could be unified with either of the two clauses corresponding to instruction 2 so exploration will *fork* and follow each of the two branches depending on the value of $R3$:

$$\begin{aligned} attacker(R1) &\rightarrow state(3, (R1, 1, 0, 1)). \\ attacker(R1) \wedge attacker(R3) \wedge R3 \neq 0 &\rightarrow state(3, (R1, 0, R3 - 1, R3)). \end{aligned} \quad (8)$$

We could make a parallel between this behavior and symbolic execution: in symbolic execution input variables that the attacker can control are marked as symbolic and a symbolic execution engine executes the program, forwarding and constraining symbolic values along the different possible paths. When the execution must split according to the value of a symbolic variable, the constraints on the symbolic value for each path are remembered and two separate instances of the execution engine continue the analysis.

Our method shares some similarities: variables controlled by the attacker are used without giving them concrete values until a conditional instruction – that has been translated into two clauses – is met. The unification process then follows two different paths where premises have been added that constrain the value of the variable.

5.4 What Is Lacking

We present here the drawbacks of this method. Some are inherent to the method, some would require more or less work to correct them.

Working with Concrete Types. For the method to be efficient, the number of instructions generating multiple Horn clauses, and the number of clauses generated for each of such instruction should remain small. However ProVerif has no semantic for concrete types (such as bit vectors or even numbers) and it is up to the user to model them. But this modeling is not obvious in ProVerif. Indeed, the definition of functions such as the addition of two bit vectors can be done either

- by constructors, that construct *new* values so it wouldn't be possible to express for instance that $1 + 0 = 1$

– or by destructors, that don't allow recursive definition, so we would need to explicitly give the result for each possible addition. In this case, if we have an instruction `add r2, r3` where `r2` and `r3` are controlled by the attacker and can take either of n and m values respectively, this instruction will be translated into $n.m$ Horn clauses which will considerably increase the complexity of the analysis.

An efficient representation of numbers should enable a translation of one instruction into only one clause (except for conditional instructions). We could imagine modifying ProVerif to add a new type of function which would have no semantic for ProVerif. When trying to unify clauses, instead of simply looking for clauses with a conclusion and a premise that a substitution could make equal, it would call a SMT solver (as it is done by symbolic execution engines). If the solver could find a constraint over the symbolic variables that enables unification, it would add the constraint to the premises of the newly generated clause.

We haven't implemented this modification, and so haven't been able to complete our analysis of the SMART architecture. However we believe it would be possible and could benefit other applications, for instance, protocols that compute arithmetic expressions.

Working with Machine Code. Our goal was to be able to model complex attack scenarios that would take advantage of the concrete representation of data and code. While having an instruction-accurate model is a first step, we are still not working on a low enough level to model attacks such as Return Oriented Programming, which would require a representation that preserves the dual semantic of bit vectors and instructions.

State Explosion. Exploring the entire state space of the program without performing substantial abstraction naturally entails serious complexity concerns. For this reason we first target small and critical designs, for which formal security proof is needed.

6 Conclusion

In this paper we presented our vision of the required properties of a formal verification method targeting hardware/software co-designs. The field of our analysis clearly differs from traditional software analysis. On the one hand, constraints about the scaling of the analysis are relaxed since we focus on small, critical part of the software, on the other hand, low-level security concerns and hardware customizations require an accurate formal representation and limit the possibilities for abstractions. After surveying different methods applied to hardware/software co-designs, we presented a different approach based on ProVerif that answer part of our requirements.

We summarize in Table 2 the adequacy of academic and industrial tools to the ideal method described in the beginning of the paper. It is interesting to

note that some tools can take as input abstract models and thus can analyze both hardware and software. However, they require to manually translate the hardware or software into an abstract model before analyzing it. Also note that ZeBu, Seamless, and SoC Designer Plus were included even if they are emulators and not formal verification tools.

Table 2. Comparison of verification tools.

Tool	Security-oriented	Type of prop.	Soundness	HW/SW modelling
NuSMV	no	CTL,LTL	sound	abstract model
UPPAAL	no	TCTL	sound	abstract model
BLAST	yes	safety	sound	software
Vis	no	CTL	sound	hardware
KLEE, FIE	yes	safety	sound	software
S2E	yes	safety	sound	software
ZeBu, Seamless, SoC Designer Plus	no	—	unsound	HW/SW

As perspectives for future work, we wish to explore other solutions such as integrating software as part of the hardware representation and maybe adding to ProVerif the ability to work with concrete types by creating an interface that one could use to plug a SMT solver such as Z3 in. We'll also consider automating the process of integrating hardware customization into the ProVerif model.

This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference #ANR-11-LABX-0031-01. Finally, we would like to express our gratitude to Bruno Blanchet for his precious help and patient contribution to our understanding of ProVerif, and to our anonymous reviewers for their insightful comments.

References

1. X. Allamigeon and B. Blanchet. Reconstruction of Attacks against Cryptographic Protocols. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, 2005.
2. L. Apvrille and Y. Roudier. SysML-Sec: A SysML Environment for the Design and Development of Secure Embedded Systems. In *APCOSEC 2013*, 2013.
3. O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFIX: Hardware-Assisted Flow Integrity Extension. In *52nd Design Automation Conference (DAC)*, 2015.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*. 1999.
5. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, 2001.

6. B. Blanchet, B. Smyth, and V. Cheval. Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2015.
7. D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.
8. E. Clarke and E. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, 1982.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*. 2000.
10. L.A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *Software Engineering, IEEE Transactions on*, 1976.
11. J.-L. Danger, S. Guilley, T. Porteboeuf, F. Praden, and M. Timbert. HCODE: Hardware-Enhanced Real-Time CFI. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, 2014.
12. D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
13. D. Dolev and Andrew C. Yao. On the Security of Public Key Protocols. *Information Theory, IEEE Transactions on*, 1983.
14. Thomas Dullien and Sebastian Porst. REIL : A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis. 2009.
15. K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego*, 2012.
16. A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving*. 2010.
17. S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. A Case of System-level Hardware/Software Co-design and Co-verification of a Commodity Multi-processor System with Custom Hardware. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2012.
18. J. King. Symbolic Execution and Program Testing. *Commun. ACM*, 1976.
19. P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.
20. D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, 2005.
21. J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I Verbauwhede, and F. Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
22. J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, 1982.
23. P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.