Journal of Cryptographic Engineering manuscript No.
(will be inserted by the editor)

SMASHUP: a toolchain for unified verification of hardware/software co-designs

Florian Lugou · Ludovic Apvrille · Aurélien Francillon

Received: date / Accepted: date

Abstract Critical and privacy-sensitive 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

Florian Lugou
Télécom ParisTech, Université Paris-Saclay, 06410, Sophia Antipolis, France
Tel.: +334-93-008407
E-mail: florian.lugou@telecom-paristech.fr

Ludovic Apvrille
Télécom ParisTech, Université Paris-Saclay, 06410, Sophia Antipolis, France
Tel.: +334-93-008406
E-mail: ludovic.apvrille@telecom-paristech.fr

Aurélien Francillon
EURECOM, Campus SophiaTech, 06410, Sophia Antipolis, France
Tel.: +334-93-008119
E-mail: aurelien.francillon@eurecom.fr

1 Acknowledgements

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.

2 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 guarantee of security.

It is 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—do not use general-purpose CPU but prefer ASIPs created with tools like Processor Designer from Synopsys or Xtensa from Cadence. In particular, research topics have recently shown great interest in hardware-assisted security solutions [3, 13, 22, 24]. 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 considered representative of many other hardware-assisted security solutions which would greatly benefit from for-

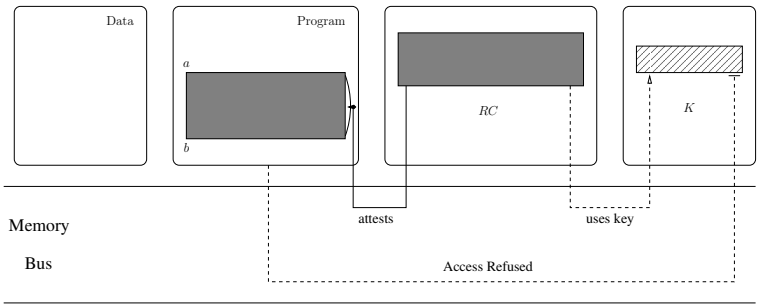

Fig. 1 SMART overview.

mal verification. This design is SMART, which stands for Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust and has been presented in [18]. 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 send the result back to a remote verifier. SMART guarantees that this hash could not 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 Fig. 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 keyed-hash message authentication code (HMAC) on a memory range $[a, b]$ passed as an argument. This HMAC relies on a key $\mathcal{K}$ to prevent a compromised software from computing the hash itself.

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 $\mathcal{RC}$ does not disable interrupts before loading the key in non-protected memory? What if control jumps into 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 the non-standard hardware into account when analyzing software to prove system-level properties. To the best of our knowledge, no general methodology for unified verification has been proposed so far. 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 threefold: 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.

We first discuss related work in Sect. 3. In Sect. 4 we present the properties that an ideal methodology for the verification of hardware/software co-designs should have. Then, in Sect. 5 and 6, we survey existing techniques. In Sect. 7, we present our conceptual approach to the problem based on ProVerif, and finally, we discuss future work before concluding.

3 Related work

While both hardware verification and software verification have been actively researched since the first digital systems were born [4, 9, 16, 21], combinatorial explosion still poses challenges when it comes to integrating both in a single verification flow.

In the embedded systems industry, where designs mixing hardware and software are commonplace, development environments such as ZeBu,[1] Seamless,[2] or

[1] `http://www.synopsys.com/Tools/Verification/hardware-verification/emulation/Pages/zebu-server-asic-emulator.aspx`

[2] `http://www.mentor.com/products/fv/seamless/`

SoC Designer Plus[3] enable to perform co-simulation of hardware and software. Simulation is fundamentally different from verification since it is driven by concrete test cases and thus avoids most abstractions needed to formally verify a system. However, simulation can only provide the same level of trust as verification does when the entire input space is covered by test cases (which is unlikely to happen, even in relatively small systems).

Some academic works [20, 23, 26, 27, 28, 30, 31] have tried to achieve unified verification of hardware and software for specific designs. For instance, in [23], Kroening et al. 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. As it will be discussed in Sect. 6, this methodology can only be applied when hardware and software are loosely coupled. Globally, this method could be applied for every language that can describe both software and hardware. For instance, we could leverage HDL-to-C compilers such as Verilator[4] to link hardware and software functions together. In this case, both software and hardware would be described in C, which would allow to perform the verification of the joint system using software verification techniques. On the other side, the work presented in Sect. 7 would enable a designer to prove properties on a system even if hardware affects how software is executed.

4 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.

4.1 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.

[3] `http://www.carbondesignsystems.com/soc-designer-plus`

[4] `http://www.veripool.org/wiki/verilator`

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.

4.1.1 Natural expression of security properties

First, designers would like to express the property they want to prove on their design as directly as possible. Translating the expected property into a combination of properties manageable by the solution but whose meanings are hard to grasp—typically formulae in conjonctive normal form—is a source of errors. The verifier would thus be more interested by solutions that can naturally handle properties such as secrecy properties or taint propagation properties.

4.1.2 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 ([15]), 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.

4.1.3 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 redesigned. 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.

4.2 Soundness of the verification algorithm

Many hardware/software co-designs provide core features that are critical either for the proper functioning 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 complete verification out. 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 do not affect the soundness of the verification.

4.3 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 [14, 29]. 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 to which extent each method can cope with hardware modification.

5 Successive verification of hardware and software

The 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. We will call these methods *successive verification* since the verification takes place in two steps. The two steps are explicitly or implicitly linked by the designer that provides a formal semantic at the junction of hardware and software.

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.

5.1 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 does not affect the core processor but is a peripheral (such as an MMU or a sensor), and the software part is handling the communication with this peripheral. In the first case, the software analysis tool—which assumes a particular semantic of the instruction set and the execution engine—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. In this case, the hardware specificities do not question the *software* abstraction made by traditional verification tools.

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 such as Vis,[5] NuSMV,[6] Incisive[7] or Formality.[8]

5.2 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 machine code. Therefore, we are mainly interested in software verification tools that can take machine code as input.

Higher-level concepts such as arrays, objects, functions, or types are not available when using assembly code. Losing such concepts means that we can't benefit from the semantic of coherent objects that the designer manually provided. For instance, it is 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 analyze 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) and of the software we want to verify.

[5] http://vlsi.colorado.edu/~vis/

[6] http://nusmv.fbk.eu/

[7] http://www.cadence.com/products/fv/enterprise_simulator/pages/default.aspx

[8] http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx

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 made lately in this direction [19, 14]. Once this formal model has been found, traditional software analysis methods like model checking [25] and bounded model checking [5] or symbolic execution [11] may be applied.

5.3 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 should not be prohibitive.

One possibility would be to apply the CounterExample Guided Abstraction Refinement methodology (described in [10]). Here, a 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.

6 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 [23, 20].

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

6.1 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 (such as memory mapped port or signals), the duality hardware/software is not relevant anymore. Conceptually, both hardware and software parts provide abstract functionalities that are described in two unrelated languages. It becomes thus possible to verify both at the same time, but still keep them distinct. As presented in Sect. 3, a good example of such verification on loosely coupled hardware and software can be found in [23].

6.2 Tightly coupled hardware and software

Hardware-based protection against software vulnerabilities often affects how the processor interprets machine code, by detecting policy violation as in [18], 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 cannot be done as in the previous section.

6.2.1 Including software as part of the hardware representation

In such cases, 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 do not want to create a formal model of the processor—which usually is an intricate manual process. We cannot 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 verifying the combined system as a whole.

6.2.2 Proving properties on the whole design

Once software has been integrated into the hardware representation of the design, traditional hardware verification tools 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. For instance, symbolic model checking attempts to reduce the state space by finding *good* abstractions that would lead to a reasonably small model. In our case, such abstractions would most likely relate software-level objects together, which would have disappeared in the combined system. For instance, let us 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*

Table 1 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

value of variable v". The meaning of s and v, however, is no longer present at the hardware level, and automatically reconstructing these objects, by predicate abstraction for instance, would be difficult. Indeed, there is no hardware concept of what a string is, nor what smaller means. For this reason such a verification scheme would probably be limited to very small programs.

6.3 Adequacy of existing tools

We summarize in Table 1 the adequacy of academic and industrial tools to the requirements described in Sect. 4. It is interesting to note that some tools can take abstract models as input and thus can analyze both hardware and software. However, using these tools requires the designer to first manually provide an abstract model corresponding to the system to verify. Also note that ZeBu, Seamless, and SoC Designer Plus were included even if they are emulators and not formal verification tools.

7 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 modeling the whole protocol should not add too much work.

7.1 Motivations for using ProVerif

ProVerif [7] is a tool for analyzing protocols. It focuses on security protocols but the generic language (pi calculus) used for ProVerif specifications and the simple reasoning of the tool, based on Horn clauses make 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 the fact that it was originally designed for a different field of applications.

7.1.1 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 [15]. 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.

7.1.2 A simple reasoning

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

Table 2 A ProVerif process and the corresponding Horn clauses.

ProVerif Process	Set of Horn Clauses

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

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

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.

7.1.3 Trace 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.

7.2 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 about the specified protocols. This will help us explain how this is done for our model and compare the performance with more traditional techniques.

7.2.1 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 2. 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.

7.2.2 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.

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 [6]—is going to be exposed.

The idea behind clause unification is to progressively expand the knowledge of the attacker. Let us 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.

7.3 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 $\mathcal{K}$ cannot 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.

7.3.1 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 into the middle of the routine or dynamic control-flow graph with indirect jumps for instance. 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 [17] or the BAP intermediate language [8] and use the already existing front ends to compile either assembly code or binary code to this intermediate representation.

An instruction at virtual address i is modeled as a process, which is translated by ProVerif into a Horn clause: $state(i, R, MEM) \rightarrow state(PC', R', MEM')$, in which 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. Therefore, 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—does not lead to $attacker(R)$ or $attacker(MEM)$, and reciprocally that the attacker cannot 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 the ProVerif operator `!`—so that the instruction could be invoked many times (in case of loops for instance). We wrote an open-source Python program named SMASHUP[9] (*Simple Modeling and Attestation of Software and Hardware Using ProVerif*) that automates the process of translating MSP430 assembly code into a set of such processes.

[9] Available at `https://gitlab.eurecom.fr/Aishuu/smashup`

7.3.2 Parallel with symbolic execution

The algorithmic efficiency of ProVerif resides in its ability to derive the complete knowledge of the attacker with as few clause 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:

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

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, the solving algorithm of 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:

$$\begin{aligned}
&attacker(R1) \wedge attacker(R2) \wedge attacker(R3) \\
&\qquad \rightarrow state(1, (R1, R2, R3, 0)).
\end{aligned} \tag{6}$$

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:

$$\begin{aligned}
&attacker(R1) \wedge attacker(R2) \wedge attacker(R3) \\
&\qquad \rightarrow state(2, (R1, R2, R3, R3)).
\end{aligned} \tag{7}$$

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 $R3$:

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

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.

7.3.3 The hardware part

The instruction-accurate description of software presented above enables us—to some extent—to bring hardware customization into the verification process. To do that, we designed SMASHUP in a modular way. Each module would represent a hardware specificity, such as an interrupt controller or a MMU. SMASHUP's users can select standard modules that we provide with the tool, or create their own.

Concretely, each module is described as a Python class that extends a common class. Modules can add predicates to the state of the system (like PC, R or MEM described above) that can be used and updated by instructions. For instance, an interrupt controller could add a register to remember whether interrupts are enabled or disabled. Also, a module can add new instructions and provide a corresponding *implementation* for each of them. This implementation describes how the instruction impacts the state of the system and the knowledge of the attacker.

The designer can then provide a high-level description of its architecture by instantiating different modules. Since the tool has not yet fully matured, this description is kept simple. Improving it could add a lot of value to SMASHUP and is left as future work.

7.3.4 Example and performance evaluation

We illustrate the capability of SMASHUP to find vulnerabilities coming from both software and hardware specificities through an example of a simple system composed of a hardware part and a software part. This example consists of a hardware and a software model that can be downloaded together with the SMASHUP tool. The former one is described in Listing 1. It only features an execution core and a standard memory.

Listing 1 Hardware Description of the System

```
1  CPU (
2      width: 4,
3      rcount: 5
4  )
5
6  MEMORY (
7      width: 4,
8  )
9
10 # uncomment the following module to test
11 # interrupts
12 # INT_UNIT ()
```

The software part described in Listing 2 shows that a secret (initially stored in register `r1`) is written in memory a certain amount of time (controlled by `r3`). Then the secret is cleared from all the addresses where it was stored.

Listing 2 Software Description of the System

```
1  .section .do_mac.call,"ax"
2      mov.w       #0x0000,    r4
3  l0:
4      cmp         r3,         r4
5      jeq         l1
6      mov.w       r1,         @r4
7      add.w       #1,         r4
8      jmp         l0
9  l1:
10     mov.w       #0x0000,    r4
11     mov.w       #0x0000,    r1
12 ; uncomment the following instruction to test
13 ; integer overflow
14 ;    add.w       #1,         r3
15 l2:
16     cmp         r3,         r4
17     jeq         l3
18     mov.w       #0x0000,    @r4
19     add.w       #1,         r4
20     jmp         l2
21 l3:
22     nop
```

With this description, we can use SMASHUP and ProVerif to assert that the secret is not leaked:

```
$ ./smashup.py -o test.pv -s .do_mac.call \
$   --clauses examples/test.s43
$ time proverif test.pv
[...]
Starting query not attacker(secret[])
RESULT not attacker(secret[]) is true.
proverif test.pv  0,22s user 0,00s system
```

The output of ProVerif match the expected answer: The secret is not leaked. We can then modify the system to model the presence of interruptions. In such case, the secret is leaked when an attacker interrupts the routine just after the secret was written to memory. The memory range that is cleared after the secret has been written to memory can also be extended by adding a software instruction (line 14) to increase the size of the memory range to be cleared by one unit. In this case, an integer overflow could happen if `r3` contains the maximum integer value and the secret would not be cleared from memory at all. This is confirmed by SMASHUP and ProVerif:

```
$ ./smashup.py -o test.pv -s .do_mac.call \
$   --clauses examples/test.s43
$ time proverif test.pv
[...]
Starting query not attacker(secret[])
[...]
The attacker has the message secret.
A trace has been found.
RESULT not attacker(secret[]) is false.
proverif test.pv  0,27s user 0,00s system
```

7.3.5 Discussion about the soundness and completeness of the approach

As mentioned in Sect. 7.1.3, ProVerif can output three results for each queried property: Either the property is proved true or false, or ProVerif is not able to prove or disprove the property. With respect to the different kinds of queries that ProVerif accepts as input, ProVerif has been proved (in [6]) to be sound but not complete. This means that, although ProVerif may fail to output either a *true* or *false* result, when it does so, then the result is valid on the input model.

We strive to keep this correctness property in our approach. However, since the hardware description of the system impacts how software is translated to Horn clauses, proving correctness on SMASHUP would require to formally define how these hardware modifications impact the translation process, which is left for future work.

However, when no hardware unit is added to the hardware model, software is translated in a straightforward way (as described in Sect. 7.3.1): Each instruction results in one or two Horn clauses. Each Horn clause describes the effect of an instruction in a particular environment. Since no side effect happens (the environment is entirely described in the *state* predicate), the correctness of the proof performed by SMASHUP and ProVerif for default hardware is easily inferred from the correctness of ProVerif.

7.4 Limitations and future work

We present here a few limitations of this method. Some are inherent to the method, some could be improved in future works.

7.4.1 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 in two ways.

- Either by constructors that construct *new* values so it would not be possible to express for instance that $1 + 0 = 1$.
- Or by destructors, which do not 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, ProVerif would call an SMT solver (as it is done by symbolic execution engines). If the solver could find an assignment of the symbolic variables that enables unification, it would add the constraint to the premises of the newly generated clause.

Implementing this modification could be part of future work. We believe it would benefit other applications, for instance, protocols that compute arithmetic expressions.

Another solution would be to find a language that shares some of the expected features presented in Sect. 7.1: an interesting attacker model, a simple reasoning and trace reconstruction. The closest language that comes to our mind is Prolog [12]. Like ProVerif, Prolog enables users to provide clauses to relate predicates. Unlike ProVerif, it is able to work with numbers and to deal with recursive definition of compound terms. Both of these would be interesting for our setup. However, Prolog does not target specifically security properties. As such, trace reconstruction and security properties—not only confidentiality, but also authenticity—may need to be pre-processed to fit our purpose. Another important difference between ProVerif and Prolog is that the former tries to unify clauses to expand the knowledge of the attacker, whereas the latter unifies clauses that can result in the required property. This means that in a simple setup, Prolog would start unifying clauses from the end of the software. If the program contains an indirect jump which can only point to two locations, a backward analysis of the program would need to consider at each step that the indirect jump could potentially target the current instruction. Evaluating the consequence of this process on performance is left for future work.

7.4.2 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 sufficiently low level to model attacks such as return-oriented programming, which would require a representation that preserves the dual semantic of bit vectors and instructions.

7.4.3 Reconstructing attack traces

As ProVerif is able to output a trace leading to a violation of a required property, 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. This could be valuable for the designer to distinguish between valid attacks and spurious traces.

8 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 a 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 addresses part of the challenges presented in the first sections.

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 integrate a SMT solver such as Z3. We will also consider automating the process of integrating hardware customization into the ProVerif model.

References

1. Allamigeon X, Blanchet B (2005) Reconstruction of Attacks against Cryptographic Protocols. In: Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop
2. Apvrille L, Roudier Y (2013) SysML-Sec: A SysML Environment for the Design and Development of Secure Embedded Systems. In: APCOSEC 2013
3. Arias O, Davi L, Hanreich M, Jin Y, Koeberl P, Paul D, Sadeghi AR, Sullivan D (2015) HAFIX: Hardware-Assisted Flow Integrity Extension. In: 52nd Design Automation Conference (DAC)
4. Armstrong RC, Punnoose RJ, Wong MH, Mayo JR (2014) Survey of existing tools for formal verification. Tech. rep., Sandia National Laboratories
5. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic Model Checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems
6. Blanchet B (2001) An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE
7. Blanchet B, Smyth B, Cheval V (2015) Automatic Cryptographic Protocol Verifier, User Manual and Tutorial
8. Brumley D, Jager I, Avgerinos T, Schwartz E (2011) BAP: A Binary Analysis Platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification
9. Camurati P, Prinetto P (1988) Formal verification of hardware correctness: introduction and survey of current research. Computer
10. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-Guided Abstraction Refinement. In: Computer Aided Verification
11. Clarke L (1976) A System to Generate Test Data and Symbolically Execute Programs. Software Engineering, IEEE Transactions on
12. Clocksin W, Mellish CS (2003) Programming in PROLOG. Springer Science & Business Media
13. Danger JL, Guilley S, Porteboeuf T, Praden F, Timbert M (2014) HCODE: Hardware-Enhanced Real-Time CFI. In: Proceedings of the 4th Program Protection and Reverse Engineering Workshop
14. Davidson D, Moench B, Ristenpart T, Jha S (2013) FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In: Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)
15. Dolev D, Yao AC (1983) On the Security of Public Key Protocols. IEEE Transactions on Information Theory
16. D'Silva V, Kroening D, Weissenbacher G (2008) A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems
17. Dullien T, Porst S (2009) REIL : A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis
18. El Defrawy K, Francillon A, Perito D, Tsudik G (2012) 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
19. Fox A, Myreen M (2010) A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In: Interactive Theorem Proving
20. Hong S, Oguntebi T, Casper J, Bronson N, Kozyrakis C, Olukotun K (2012) 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
21. Kern C, Greenstreet MR (1999) Formal verification in hardware design: A survey. ACM Trans Des Autom Electron Syst
22. Koeberl P, Schulz S, Sadeghi AR, Varadharajan V (2014) TrustLite: A Security Architecture for Tiny Embedded Devices. In: Proceedings of the Ninth European Conference on Computer Systems
23. Kroening D, Sharygina N (2005) 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
24. Noorman J, Agten P, Daniels W, Strackx R, Van Herrewege A, Huygens C, Preneel B, Verbauwhede I, Piessens F (2013) 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)

25. Queille JP, Sifakis J (1982) Specification and Verification of Concurrent Systems in CESAR. In: Proceedings of the 5th Colloquium on International Symposium on Programming
26. Schlich B (2010) Model checking of software for microcontrollers. ACM Trans Embed Comput Syst
27. Schmidt B, Villarraga C, Fehmel T, Bormann J, Wedler M, Nguyen M, Stoffel D, Kunz W (2013) A new formal verification approach for hardware-dependent embedded system software. IPSJ Transactions on System LSI Design Methodology
28. Semeria L, Ghosh A (2000) Methodology for hardware/software co-verification in c/c++. In: Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific
29. Subramanyan P, Arora D (2014) Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In: Design, Automation and Test in Europe Conference and Exhibition (DATE)
30. Villarraga C, Schmidt B, Bormann J, Bartsch C, Stoffel D, Kunz W (2013) An equivalence checker for hardware-dependent embedded system software. In: Formal Methods and Models for Codesign (MEMOCODE), 2013 Eleventh IEEE/ACM International Conference on
31. Villarraga C, Schmidt B, Bao B, Raman R, Bartsch C, Fehmel T, Stoffel D, Kunz W (2014) Software in a hardware view: New models for hw-dependent software in soc verification and test. In: 2014 International Test Conference