Verifying Information Flow Properties of Firmware using Symbolic Execution

Pramod Subramanyan Sharad Malik
Dept. of Electrical Engineering, Princeton University.

Hareesh Khattri Abhranil Maiti Jason Fung
Security Center of Excellence, Intel Corporation.

***Abstract*—Verifying security requirements of the firmware in contemporary system-on-chip (SoC) designs is a critical challenge. There are two main difficulties in addressing this problem. Security properties like confidentiality and integrity cannot be specified with commonly-used property specification schemes like assertion-based verification/linear temporal logic (LTL). Second, firmware interacts closely with other hardware and firmware which may be untrusted/malicious and their behavior has to be correctly modelled for the verification to be sound and complete.**

In this paper, we propose an approach to verify firmware security properties using symbolic execution. We introduce a property specification language for information flow properties of firmware which intuitively captures the requirements of confidentiality and integrity. We also propose an algorithm based on symbolic execution to verify these properties. Evaluation on a commercial SoC design uncovered a complex security bug missed by simulation-based testing.

I. Introduction

Recent years have seen a proliferation of *firmware* for implementing key hardware platform functions. This is because firmware enables rapid development and the ability to deploy bugfixes after the system is released. Unfortunately, it also opens up another attack vector for security breaches. Indeed, several high-profile security bugs have been found in the firmware of devices as diverse as cars [19], wireless routers [11, 13] and personal computers [5, 17]. Therefore, a critical problem facing the semiconductor industry today is ensuring that firmware meets its security requirements.

A. Challenges in Firmware Security Verification

Firmware verification is different from software verification because firmware interacts closely with hardware and other firmware. These interacting components have to be modeled correctly for the verification to be sound. Security verification is especially hard because detecting vulnerabilities requires reasoning about worst-case scenarios and *all* possible states/inputs of the system, not just commonly-used states and legal inputs. Constructing such detailed models in a sound manner using, for example, iterative abstraction refinement is extremely time consuming and is at odds with the time-to-market pressures imposed by the competitive environment.

Traditional property specification schemes such as assertion-based verification and linear temporal logic (LTL) cannot express security requirements like confidentiality and availability which require reasoning about information flow. Confidentiality and integrity can be intuitively specified using *information flow* properties and one method for verifying information flow properties is through dynamic taint analysis [2, 9, 10, 18, 22, 24]. Dynamic taint analysis (DTA) associates a "taint bit" with each object in the program or word in memory. Taint propagation rules then set the taint bit of the output of a computation if its input(s) are also tainted. Confidentiality violations can be detected by tainting the secret and raising an error when the taint propagates to an untrusted object/memory location. Integrity violations are detected by tainting untrusted objects and raising an error when the taint propagates to a trusted location.

While DTA enables intuitive property *specification*, it has deficiencies in the *verification* aspect. DTA can determine if the property has been violated given an instruction sequence/trace. However, since it is a *dynamic* analysis, it cannot be used to search over the space of all possible instruction sequences to exhaustively prove the absence of property violations. Secondly, creating appropriate taint propagation rules is challenging due to the problems of under- and over-tainting [2, 18]. Over-tainting can result in a deluge of false-positives [18] while under-tainting can miss bugs.

Static taint analysis can verify information flow properties in software. However, current static taint analysis techniques are based on programming languages with secure type systems [20, 21]. These techniques cannot be applied on existing firmware because significant parts are written in assembly language necessitating analysis at the level of binary code. Furthermore, it is not possible to capture security requirements posed by hardware/firmware interactions with these methods.

B. Our Contributions

This paper introduces a technique for verifying firmware security properties using symbolic execution. Our first contribution is a property specification language for information flow properties of firmware. Firmware information flow properties are specified as a tuple consisting of the following components: source, destination, source predicate and destination predicate. The property specifies that information cannot "flow" from source to destination when the respective predicates are valid. Confidentiality can be verified with a property where the source is a secret and the destination is any untrusted location. Similarly, integrity can be verified with a property where the source is any untrusted location while the destination is a sensitive firmware register. To the best of our knowledge, ours is the first property specification scheme for confidentiality/integrity properties in *firmware*.[1]

Our second contribution is an algorithm based on symbolic execution to verify information flow properties. The algorithm exhaustively explores all paths in the program and creates

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. It was performed during the first author's internship at the Security Center of Excellence, Intel Corp.

[1]Note specification languages do exist for hardware-only properties [1]

symbolic expressions corresponding to the computation along each path. It then uses a constraint solver to check whether there exist some two different values at the source which can result in different values for the destination. We also show how selective symbolic execution can be used to model the side-effects of memory-mapped I/O accesses. The novel aspects are: (i) extension of symbolic execution to prove information flow properties; and (ii) the modeling of side-effects to MMIO accesses. Note existing symbolic simulators [3, 6, 7, 12] can verify safety properties but *not* information flow.

Our algorithm explores all control-flow paths and all values for each variable. As a result it is more precise than DTA: under-tainting is mitigated and over-tainting is eliminated. Our algorithm analyzes binary code and does not require firmware to be written in a specific programming language. Hence, it is well-suited for analyzing real-world firmware.

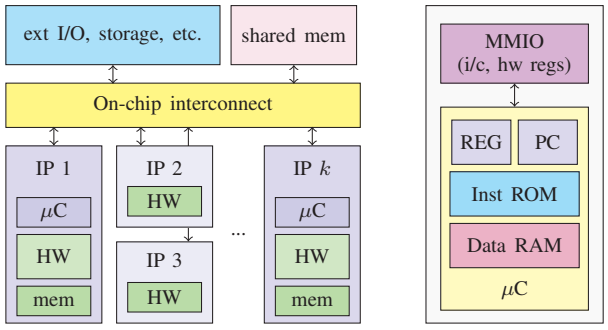
We evaluate the proposed approach on firmware from an upcoming commercial SoC design. The firmware had previously undergone both simulation-based testing and manual code review and yet the symbolic execution engine identified a tricky firmware security bug that was missed by the previous validation efforts. Scalability of the algorithm is promising; non-trivial real-world firmware, of size up to a thousand instructions, can be analyzed.

II. System and Threat Model Overview

This section provides a brief overview of the system model and threat model considered in this work.

A. System-On-Chip Model

Fig. 1: System Model Overview



(a) System-on-Chip Overview (b) FW view of SoC

A pictorial overview of the high-level architecture of the SoC design is shown in Figure 1a. The SoC consists of a set of interacting *IPs*, a shared I/O space, an on-chip interconnect and possibly a shared memory. Generally, each IP consists of a microcontroller, specialized hardware components and private memories. The firmware executes on the microcontroller and interacts with the specialized hardware in its own IP as well as other IPs through memory-mapped I/O (MMIO).

Some of the simpler IPs may not have a microcontroller or private memories, but they too interact with firmware in the other IPs through MMIO and over the on-chip interconnect. The above is a general description of SoC architecture and applies to many SoCs, see for instance, [23].

1) Firmware View of System: The firmware's view of the system is shown in Figure 1b. It consists of a set of architectural registers, a special register known as the program counter, an instruction ROM which contains the firmware code and a separate data RAM. This considers single-threaded firmware, however other hardware and firmware interactions are modeled and execute concurrently with the firmware thread.

B. Threat Model

The verification problem is tackled in a modular manner. Each IP is verified separately and so the threat model is defined from the perspective of the individual IPs.

1) Identifying the Trust Boundary: For each IP we identify the other IPs which are its *trust boundary*. The trust boundary is the subset of other IPs this IP fully trusts. For example, an IP involved in security critical functionality such as secure boot, will likely *not trust* the camera and GPS IPs. Therefore, these IPs *will be outside* its trust boundary and any inputs it receives from these IPs are untrusted. Keeping the trust boundary small helps keep the attack surface small.

2) Security Objectives: The two classes of security objectives we consider are *confidentiality* and *integrity* of firmware assets. We wish to keep firmware secrets, *e.g.*, encryption keys, confidential from untrusted IPs. Similarly, we wish to preserve the integrity of firmware assets. For example, we wish to ensure the integrity of firmware control-flow by ruling out stack smashing/buffer overflow attacks in the presence of arbitrary inputs from untrusted IPs. We also wish to ensure that untrusted IPs cannot modify the value of sensitive control/configuration registers, such as memory protection configuration registers. In this work, we do not consider other security requirements like availability and side channel attacks.

3) Modelling the Attacker: We assume that memory, I/O and hardware registers controlled by untrusted IPs contain arbitrary values. In other words, outputs of an untrusted IP are *unconstrained* so reads from these locations return arbitrary values. Similarly, untrusted IPs may send invalid commands in an attempt to exploit, for example, buffer overflow bugs.

III. Formal Model

We now describe the firmware state and execution model.

A. Execution State

Firmware state is modeled using the following components.

$\mathcal{R}$ is a set of bit-vector variables which represent the architectural registers of the microcontroller. This includes all registers including the accumulator, flag register, segment registers and stack pointer.

$\mathcal{M}$ is the data memory which is a map from bit-vectors of size $\mathtt{A_W}$ to bit-vectors of size $\mathtt{M_W}$. $2^{\mathtt{A_W}}$ is the size of the address space and $\mathtt{M_W}$ is the size of the smallest addressable memory location (usually 8 bits). $\mathcal{M}[a]$ returns the value at memory address a. $\cup$ denotes the write/update operation: in $\mathcal{M}' := \mathcal{M} \cup [a \mapsto v]$, $\mathcal{M}'$ is the same memory as $\mathcal{M}$ except $\mathcal{M}'[a] = v$, *i.e.*, memory address a now contains value v.

$\mathcal{I}$ models the instruction ROM. $\mathcal{PC}$ is the address of the current instruction. $\mathcal{I}[\mathcal{PC}]$ is the currently executing opcode.

A set of bit-vector variables $\mathcal{X}$=$\{memop,\ memaddr,\ datain,\ dataout\}$ contain information about the current memory operation. $memop = \mathsf{NOP}$ means that the current instruction does not read/write from memory or MMIO, $memop = \mathsf{RD}$ denotes a read and WR is a write. $memaddr$ is the address of the current memory operation, $datain$ is the data read *from* memory (or I/O) and $dataout$ is the data being written to memory or I/O. These variables help model MMIO and accesses to untrusted memory.

Firmware state is the tuple $\mathcal{F} = (\mathcal{R}, \mathcal{PC}, \mathcal{M}, \mathcal{X})$. The initial state is $\mathtt{S}_\mathcal{F} = (\mathtt{S}_\mathcal{R}, \mathtt{S}_\mathcal{PC}, \mathtt{S}_\mathcal{M}, \mathtt{S}_\mathcal{X})$. $\mathtt{S}_\mathcal{R}$ is the initial value of registers $\mathcal{R}$; $\mathtt{S}_\mathcal{M}$ and $\mathtt{S}_\mathcal{X}$, are respectively the initial states of $\mathcal{M}$ and $\mathcal{X}$. All of these are expressions in quantifier-free first order logic using bit-vector and array datatypes [14] (referred to as `QF_ABV` in SMT-LIB).[2] $\mathtt{S}_\mathcal{PC}$ is a bit-vector constant with the initial value of the $\mathcal{PC}$.

B. Execution Model

As shown in Figure 2a, the symbolic simulator exhaustively follows all paths that are reachable from the initial state $\mathtt{S}_\mathcal{F}$ and verifies that properties hold on all of these [6, 15].

(a) Fully symbolic execution (b) Selective symbolic execution.

Fig. 2: Execution model. In (b) the shaded boxes show single-path (concrete) execution through the simulation model due to MMIO.

1) Expressions for State Update: For each state element f in $\mathcal{F}$ we define an expression $E_f(\mathcal{F})$ which specifies how this is to be updated by the execution of the opcode $\mathcal{I}[\mathcal{PC}]$. These expressions are all in `QF_ABV`. For instance, if opcode 1 increments the stack pointer, opcode 2 decrements it and all other opcodes leave it unchanged, then $E_\mathcal{SP} = \mathtt{ite}(\mathcal{I}[\mathcal{PC}] = 1,\ \mathcal{SP}+1,\ \mathtt{ite}(\mathcal{I}[\mathcal{PC}] = 2,\ \mathcal{SP}-1,\ \mathcal{SP}))$. `ite` is the if-then-else operator. The set of expressions $\vec{E}_\mathcal{R}$ extends E_f to the set $\mathcal{R}$ as follows: $\vec{E}_\mathcal{R}(\mathcal{F}) = \{E_r(\mathcal{F})$ for each $r \in \mathcal{R}\}$. $\vec{E}_\mathcal{X}$ is defined analogously over the set $\mathcal{X}$.[3]

2) Selective Symbolic Execution: Memory-mapped I/O (MMIO) poses unique challenges for symbolic execution, because a read/write from/to MMIO might have "side-effects", *e.g.*, initiating a hardware state machine. Ideally, we would model all these symbolically. In practice, constructing such a model is not feasible because it is very time-consuming [12].

[2]In other words, E_f is an expression involving unary and binary bit-vector operators such as `and`, `or`, `not`, `add`, `sub`, `concat`, `extract`, and so on as well as the `ite` (if-then-else), and memory read and update operators.

[3]The definitions of $\vec{E}_\mathcal{R}$ and $E_\mathcal{PC}$ can be extended to model asynchronous events such as interrupts. In this work, we do not consider interrupts because the microcontroller used in the evaluation does not support them.

Therefore, we use selective symbolic execution to model the MMIO side-effects. Simulation models of the SoC are usually available during SoC design and these model MMIO accesses. We use these to execute the "side-effects" of MMIO reads/writes. This means we have to convert the symbolic expressions into sets of concrete values, execute the simulator for each concrete value, and then resume symbolic execution with the simulation results. This process is shown in Figure 2b. This process of conversion from symbolic to concrete states and back is tractable for firmware because typically accesses are only made to a small number of hardware registers.

IV. Property Specification and Verification

Algorithm 1 Symbolic Execution

Inputs: $\mathtt{S}_\mathcal{F}$, *src*, *srcpred*, *dst*, *dstpred*

1: $stack.push(\langle \mathtt{S}_\mathcal{F}, \mathtt{S}_\mathcal{F}, \mathtt{true}, \mathtt{true}\rangle)$
2: **while** $\neg empty(stack)$ **do**
3: ▷ $\mathtt{C}^1_\mathcal{F}$ and $\mathtt{C}^2_\mathcal{F}$ represent the current firmware state.
4: $\langle \mathtt{C}^1_\mathcal{F}, \mathtt{C}^2_\mathcal{F}, P^1, P^2\rangle \leftarrow stack.pop()$
5:
6: $T \leftarrow P^1 \wedge P^2 \wedge dstpred(\mathtt{C}^1_\mathcal{X}) \wedge dstpred(\mathtt{C}^2_\mathcal{X})$
7: **if** $sat[T \wedge \mathtt{C}^1_\mathcal{X}.dst \neq \mathtt{C}^2_\mathcal{X}.dst]$ **then** ▷ check properties
8: **display** violation
9: **end if**
10:
11: **if** $finished(\mathtt{C}^1_\mathcal{X}, \mathtt{C}^2_\mathcal{X})$ **then** ▷ check for completion
12: **continue**
13: **end if**
14:
15: ▷ $\mathtt{N}^1_\mathcal{F}$ and $\mathtt{N}^2_\mathcal{F}$ is state after this instruction is executed.
16: $\mathtt{N}^1_\mathcal{X} \leftarrow \vec{E}_\mathcal{X}(C^1_\mathcal{F})$
17: $\mathtt{N}^2_\mathcal{X} \leftarrow \vec{E}_\mathcal{X}(C^2_\mathcal{F})$
18:
19: ▷ check for MMIO and handle it
20: **if** $isMMIO(\mathtt{N}^1_\mathcal{X})$ **then**
21: $\langle \mathtt{N}^1_\mathcal{X}, \mathtt{N}^2_\mathcal{X}\rangle \leftarrow execMMIO(\mathtt{C}^1_\mathcal{X}, \mathtt{C}^2_\mathcal{X})$
22: **end if**
23:
24: ▷ rewrite $datain$ if $memaddr$ matches the source
25: **if** $overlaps(\mathtt{N}^1_\mathcal{X}, src)$ **then**
26: $\mathtt{N}^1_\mathcal{X}.datain = \mathtt{ite}($
$overlaps(\mathtt{N}^1_\mathcal{X}.memaddr, src) \wedge srcpred(\mathtt{C}^1_\mathcal{X}),$
$newVar(), \mathtt{N}^1_\mathcal{X}.datain)$
27: $\mathtt{N}^2_\mathcal{X}.datain = \mathtt{ite}($
$overlaps(\mathtt{N}^2_\mathcal{X}.memaddr, src) \wedge srcpred(\mathtt{C}^2_\mathcal{X}),$
$newVar(), \mathtt{N}^2_\mathcal{X}.datain)$
28: **end if**
29:
30: ▷ compute next value of $\mathcal{R}$, $\mathcal{M}$
31: $\langle \mathtt{N}^1_\mathcal{R}, \mathtt{N}^2_\mathcal{R}\rangle \leftarrow \langle \vec{E}_\mathcal{R}(\mathtt{C}^1_\mathcal{X}), \vec{E}_\mathcal{R}(\mathtt{C}^2_\mathcal{X})\rangle$
32: $\langle \mathtt{N}^1_\mathcal{M}, \mathtt{N}^2_\mathcal{M}\rangle \leftarrow \langle \vec{E}_\mathcal{M}(\mathtt{C}^1_\mathcal{X}), \vec{E}_\mathcal{M}(\mathtt{C}^2_\mathcal{X})\rangle$
33:
34: ▷ find all reachable values of PC and explore them.
35: **for all** $\mathtt{PC} = E_\mathcal{PC}(\mathtt{C}^1_\mathcal{X})$ **do**
36: $(\mathtt{N}^1_\mathcal{X}.\mathcal{PC}, \mathtt{N}^1_\mathcal{X}.\mathcal{PC}) = (\mathtt{PC}, \mathtt{PC})$
37: $P^1 \leftarrow P^1 \wedge E_\mathcal{PC}(\mathtt{C}^1_\mathcal{X}) = \mathtt{PC}$
38: $P^2 \leftarrow P^2 \wedge E_\mathcal{PC}(\mathtt{C}^2_\mathcal{X}) = \mathtt{PC}$
39: $stack.push(\langle \mathtt{N}^1_\mathcal{X}, \mathtt{N}^2_\mathcal{X}, P^1, P^2\rangle)$
40: **end for**
41: **end while**

In this section, we describe the specification language for

information flow properties and show how these are verified.

A. Specifying Information Flow Properties

The property specification language for firmware security properties is based on two insights. First, almost all interesting firmware security assets, such as secret keys, sensitive configuration registers and untrusted input registers are accessed through MMIO and memory. Therefore, it would make sense to use firmware address ranges and architectural registers as first class entities in the property specification language.

Secondly, security requirements such as confidentiality and integrity are essentially statements about information flow. These express the requirement that either a firmware secret must not "flow" to an untrusted value (confidentiality), or an untrusted value must not "flow" to a sensitive asset (integrity).

The specification language allows the definition of information flow properties consisting of the following elements.

1) A src which is a range of firmware memory addresses.
2) A predicate srcpred associated with the source which specifies when the data at src is valid. For example, we may allow a register to be programmed from an input port during the boot process, but not afterwards with the predicate $\neg boot$.
3) A dst which is a member of the firmware state $\mathcal{F}$.
4) A predicate dstpred for **dst** which specifies when data at dst is valid similar to (2) above.

The property holds if data read from src when srcpred=1 never influences a value written to dst when dstpred=1. Note srcpred and dstpred are only evaluated at the time of the read and write respectively.

B. Verifying Information Flow Properties

Algorithm 1 shows how information flow properties can be verified using symbolic execution. It performs a depth-first search (DFS) of all reachable instructions and checks whether the information flow property specified by (src, dst, srcpred, dstpred) holds for all of them. The *stack* keeps track of paths to be visited and the *path constraints* P^1 and P^2 determine the conditions under which this particular path is taken.

There are two main enhancements over previous symbolic execution engines [3, 6, 7, 12]. The first is that the engine maintains *two* copies of the state in $\mathtt{C}^1_{\mathcal{X}}$ and $\mathtt{C}^2_{\mathcal{X}}$. This is so that we can functionally test whether assigning different values to *src* results in different values at dst. This check is performed in line 7. The substitution of the source with "fresh" unconstrained variables is performed in lines 25-28. The other difference is the handling of MMIO instructions in lines 20-22.

Typical use of the algorithm is as follows. $\mathtt{S}_{\mathcal{PC}}$ is set to an appropriate initial value and a target value for the $\mathcal{PC}$ is specified as a trigger to stop symbolic execution. The algorithm then symbolically explores all paths between $\mathtt{S}_{\mathcal{PC}}$ and the target. It terminates when *stack* is empty.

To understand Algorithm 1, let us consider its execution on the code shown in Figure 3.[4] The property here states that the

[4]We show the algorithm in C-like pseudocode to make understanding easier but the analysis is done on binary code.

```
#define N 2
uint8_t tbl[] = { 1, 1 };    // address of tbl = 0x100
uint8_t data = 3;            // &data=0x102
uint8_t IO_REG = 1;          // &IO_REG=0x200.

void foo(int r1) {
    if (r1 < 0 || r1 >= N) return;
    IO_REG = tbl[r1];
}
```

Fig. 3: Integrity property example: src=r1, dst=$dataout$, srcpred=`true` and dstpred=$memaddr = \texttt{0x200} \wedge memop = \mathsf{WR}$.

untrusted value `r1` must not influence the value of `IO_REG`.

Suppose, due to a typo `N`=3 instead of the correct value 2. The symbolic state computed by the algorithm when it reaches the assignment to `IO_REG` would be:

$P^1 = \neg(x^1 < 0 \vee x^1 \geq 3)$	$P^2 = \neg(x^2 < 0 \vee x^2 \geq 3)$
$dataout^1 = \mathcal{M}^1[\texttt{0x100} + x^1]$	$dataout^2 = \mathcal{M}^2[\texttt{0x100} + x^2]$
$\mathcal{M}^1 = \mathcal{M}^2 = [\texttt{0x100} \mapsto 1, \texttt{0x101} \mapsto 1, \texttt{0x102} \mapsto 3, \ldots]$	
$memaddr^1 = memaddr^2 = \texttt{0x200}$	
$memop^1 = memop^2 = \mathsf{WR}$	

In the above, x^1 and x^2 are the new variables created to represent the untrusted value `r1`. When the solver evaluates whether $dataout^1 \neq dataout^2$ is possible along with P^1, P^2 and the predicates, it will find $x^1 = 1, x^2 = 2$ and report this error. Once we fix the bug and `N`=2, then $(P^1, P^2) = (x^1 \geq 0 \wedge x^1 < 2, x^2 \geq 0 \wedge x^2 < 2)$. Now it is not possible make $dataout^1 \neq dataout^2$ while satisfying P^1 and P^2, so the algorithm will not report in an error.

This is an example where DTA fails. Under typical taint propagation policies, if the address for a memory read is tainted but the data pointed to by the address is not tainted, the result of the read is *not* tainted. Under such a policy, the bug is not detected by DTA as the taint does not propagate from `r1` to `IO_REG`. Changing the policy to taint the result of a memory read when the address is tainted results in overtainting. DTA reports a problem even when the bug is fixed and `N`=2.

Now suppose src is `data`, while dst and dstpred are the same as before. This property states that the secret value `data` must not influence untrusted register `IO_REG`. Clearly, a violation exists if `N`=3 and it will be detected by Algorithm 1.

To detect this issue, DTA needs an instruction sequence where `r1`=2 in order to expose the violation. In other words, DTA cannot detect a violation without a trace that "activates" the problem. These examples demonstrate the advantages of our technique: exhaustive analysis of all program paths and states and improved precision over DTA.

V. Evaluation

We evaluated our approach by examining part of the firmware of an upcoming commercial phone/tablet SoC. This section provides background on the SoC and the IP examined, presents the methodology and results of the evaluation.

A. Methodology

1) SoC and IP overview: The overall structure of the SoC is similar to the model presented in Figure 1a. It consists

of a number of IPs for various functions such as display, camera, touch sensing, etc. This evaluation examined a single component IP, called the PTIP. The PTIP is involved in security sensitive "flows" like secure boot. It contains a proprietary 32-bit microcontroller which executes the firmware. The firmware interacts with the other IPs in the SoC through various hardware registers accessed using MMIO.

2) Security Objectives: The PTIP firmware interacts with system software, devices drivers and other untrusted IPs. Since these entities, especially the system software and drivers, may be compromised by malware, these are all untrusted.

We explored two main security objectives as part of the evaluation. First, the PTIP memory holds a sensitive cryptographic key called the *IPKEY*. We wanted to ensure that these untrusted entities could not access the *IPKEY*. Second, we wanted to ensure control-flow integrity of the PTIP firmware.

The total size of the PTIP firmware is approximately a few tens of thousands of static instructions. Due to limited time, this evaluation focused on a set of message handler functions which send and receive commands/messages from the (untrusted) system software, drivers and other IPs. We believe these are most likely to be vulnerable because they have the "closest" interactions with the untrusted entities.

3) Methodology: We created an instruction-level abstraction (ILA) of the PTIP microcontroller using the template-based synthesis methodology from [25]. We then used the ILA to generate a symbolic execution engine using the C++ API of version 4.3.2 of the Z3 SMT solver. This symbolic execution engine was integrated with a pre-existing simulator for this microcontroller to model the MMIO reads and writes.

We used the pre-existing simulator to execute the reset phases of the firmware, and started symbolic execution using the post-reset state as the initial state. Each message handling function (along with all the functions in its call graph) was analyzed separately. The symbolic execution was run with a time limit of 30 minutes on an Intel Xeon server.

B. Properties Verified

We created three representative information flow properties to capture the security requirements mentioned above. [5]

1) Confidentiality of IPKEY: This was framed as two information flow properties. For the first property, the source was the address of *IPKEY*. The destination was $\mathcal{PC}$ and both source and destination predicates were `true`. The security requirement here states that the firmware control flow must *not* depend on the value of *IPKEY* because different paths through the firmware may have different externally visible behavior and this could leak information about the key.

For the second property, the source was again the address of *IPKEY* in memory but the destination was $dataout$. This property states that the *IPKEY* must not be written to an untrusted MMIO location.

[5]Besides information flow properties, the symbolic execution engine we developed can also verify safety properties like the symbolic simulators in [3, 7, 12]. This is helpful for functional verification and certain security properties. We do not report these results due to a lack of space.

2) Integrity of Message Handler Table: PTIP memory contains a message handler table called `MHT`. The `MHT` contains a list of pointers to functions that handle the messages received from other IPs. The security concern here is that malware could potentially attempt to rewrite the message handler table to execute malicious code.

C. Experimental Results

1) Scalability: Table I summarizes the results from our analysis. A total of five message handlers were evaluated. The size of these routines ranges from roughly 100 to 800 instructions. Note these are *static instructions*. Due to loops, some instructions will be executed more than once.

The static instruction counts were obtained after we abstracted certain functions that were irrelevant to the security properties being examined. The abstraction replaces these functions with code that "trashes" the registers used by the routine and returns immediately. Interestingly, we used symbolic execution to prove that the other registers were correctly preserved by the routine, *i.e.*, to prove the abstraction correct.

The column labelled "analyzed instructions" shows the number of dynamic instructions *analyzed* by the symbolic execution engine when exhaustively exploring all paths. The "time" column shows the execution time in seconds. The execution is killed after approximately half-an-hour. We can see from the results that *more than half a million instructions can be analyzed in about 30 minutes*, which demonstrates that the execution engine is scalable to real-world firmware sizes.

For two of the handlers, the symbolic execution engine did not finish exploring all paths. This is due to exponential blowup in the number of paths in the program *aka path explosion*. This is a problem that affects all symbolic execution based techniques, including those that do not verify information properties [3, 6, 7, 12]. Solving this problem requires identifying the loop(s) where path explosion occurs, and replacing these loops with sound abstractions. This process is conceptually straightforward, but time-consuming in practice.

2) Bug(s) Identified: The PTIP firmware had previously undergone simulation-based testing and manual code review. However, we were still able to identify a tricky security bug in Handler1 that could lead to IPKEY exposure.

The bug was caused by the firmware reading data from a table before validating the table index, which was controlled by an untrusted entity. Therefore, the untrusted entity could send an malicious invalid command that would end up reading the IPKEY and leaking information about its value to the attacker. The bug was difficult to detect because the specific command which triggered the overflow is not easy to discover through testing or manual review. The symbolic analysis performed by our tool which involves reasoning over all possible input values was essential in crafting this malicious command.

VI. Related Work

The DART [16] and KLEE [6] projects are the precursors of subsequent work in symbolic execution. They combined modern constraint solvers and dynamic analysis to generate tests

Function	static insts	IPKEY: Property 1		IPKEY: Property 2		MHT Property	
		analyzed insts	time (s)	analyzed insts	time (s)	analyzed insts	time (s)
Handler1	663	5690	5.80s	5690	6.24s	5690	5.80s
Handler2	788	534499*	1946.03s	546029*	1954.19s	534499*	1946.03s
Handler3	790	476169*	1945.71s	421323*	1897.29s	476169*	1945.71s
Handler4	213	817	0.62s	817	0.64s	817	0.62s
Handler5	95	230	0.16s	230	0.16s	230	0.16s

TABLE I: Summary of Symbolic Execution Results. The asterisk (*) indicates symbolic execution timed out.

for software programs. S^2E builds on the KLEE infrastructure and allows symbolic execution of system software. S^2E also introduced selective symbolic execution and applied it on system calls and kernel operations. We use selective symbolic execution to model MMIO reads/writes. Bazhaniuk et al. [3] use the S^2E infrastructure to verify security properties of system management mode software in x86 systems. FIE [12] also builds on the KLEE infrastructure but makes extensions for scalable symbolic execution of firmware in TI MSP430 microcontrollers. Optimizations introduced in FIE: memory smudging and state pruning are orthogonal to our work and can be implemented in our framework to improve scalability. Unlike our work, all these frameworks can verify only safety properties, *not* confidentiality or integrity.

A large body of work also exists in area of dynamic taint analysis, *e.g.*, [2, 9, 10, 18, 22, 24]. These techniques suffer from both false positives and false negatives due to the problems of under- and over-tainting. Our technique is based on symbolic analysis and does not result in false positives. An overview of DTA and symbolic execution techniques is presented in [22]. We show how symbolic execution can be used to verify information flow; this is missing from [22] which treats DTA and symbolic execution separately.

Software model checkers [4, 8] are more scalable than symbolic execution because they avoid path explosion by *implicit* path enumeration. However, current software model checkers do not support information flow properties. Extending software model checkers in order to verify information flow properties is an important area for further work.

VII. Conclusion

An important problem facing the semiconductor industry today is ensuring that the security requirements of firmware components are met. In this paper, we introduced a property specification language and verification algorithm for specifying and verifying firmware security properties. The property specification language makes it easy to specify security properties such as confidentiality and integrity. Our verification algorithm is based on symbolic execution and exhaustively explores all paths in a firmware routine and proves the absence of property violations. It is also more precise than dynamic taint analysis schemes. Our evaluation of the algorithm was conducted on the firmware of an upcoming commercial SoC. We were able to detect a tricky firmware security bug that had been missed by simulation-based testing as well as manual code review. Scalability is promising and experiments show that non-trivial real world world firmware can be analyzed.

References

[1] JasperGold: Security Path Verification App. http://www.jasper-da.com/products/jaspergold-apps/security_path_verification_app, 2015.

[2] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *Security and Cryptography*, 2013.

[3] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. R. Tuttle, and V. Zimmer. Symbolic execution for bios security. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, 2015.

[4] D. Beyer, T. A Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *Int'l Journal on Software Tools for Technology Transfer*, 9(5-6), 2007.

[5] Symantec Security Response Blog. Mac vulnerability could provide persistent and stealthy access. http://www.symantec.com/connect/blogs/mac-vulnerability-could-provide-persistent-and-stealthy-access, 2015.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation*, 2008.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Architectural Support for Programming Languages and Operating Systems*, 2011.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2004.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Symposium on Operating Systems Principles*, 2005.

[10] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *IEEE/ACM Int'l Symposium on Microarchitecture*, 2004.

[11] Z. Cutlip. Netgear Root Compromise via Command Injection. http://shadow-file.blogspot.co.uk/2013/10/netgear-root-compromise-via-command.html, 2013.

[12] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

[13] J. J. Drake. ASUS Router infosvr UDP Broadcast root Command Execution. https://github.com/jduck/asus-cmd/blob/master/README.md, 2014.

[14] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Computer Aided Verification*, 2007.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Programming Language Design and Implementation*, 2005.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[17] T. Hudson, X. Kovah, and C. Kallenberg. ThunderStrike 2: Sith Strike. In *Black Hat USA Briefings*, 2015.

[18] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Network and Distributed System Security Symposium*, 2011.

[19] C. Miller and C. Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. In *Black Hat USA Briefings*, 2015.

[20] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Principles of Programming Languages*, 1999.

[21] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Selected Areas in Communications*, 2003.

[22] E.J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Security and Privacy*, 2010.

[23] R. Sinha, P. Roop, and S. Basu. The AMBA SOC Platform. In *Correct-by-Construction Approaches for SoC Design*. Springer New York, 2014.

[24] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information Systems Security*, 2008.

[25] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based Synthesis of Instruction-Level Abstractions for SoC Verification. In *Formal Methods in Computer-Aided Design*, 2015.

A. Disclosure

This research was evaluated using one commercial SoC design. It needs more testing before full deployment in commercial environments.