Inception: System-Wide Security Testing of Real-World Embedded Systems Software

Nassim Corteggiani
Maxim Integrated and EURECOM

Giovanni Camurati
EURECOM

Aurélien Francillon
EURECOM

Abstract

Connected embedded systems are becoming widely deployed, and their security is a serious concern. Current techniques for security testing of embedded software rely either on source code or on binaries. Detecting vulnerabilities by testing binary code is harder, because source code semantics are lost. Unfortunately, in embedded systems, high-level source code (C/C++) is often mixed with hand-written assembly, which cannot be directly handled by current source-based tools.

In this paper we introduce Inception, a framework to perform security testing of complete real-world embedded firmware. Inception introduces novel techniques for symbolic execution in embedded systems. In particular, *Inception Translator* generates and merges LLVM bitcode from high-level source code, hand-written assembly, binary libraries, and part of the processor hardware behavior. This design reduces differences with real execution as well as the manual effort. The source code semantics are preserved, improving the effectiveness of security checks. *Inception Symbolic Virtual Machine*, based on KLEE, performs symbolic execution, using several strategies to handle different levels of memory abstractions, interaction with peripherals, and interrupts. Finally, the *Inception Debugger* is a high-performance JTAG debugger which performs redirection of memory accesses to the real hardware.

We first validate our implementation using 53000 tests comparing Inception's execution to concrete execution on an Arm Cortex-M3 chip. We then show Inception's advantages on a benchmark made of 1624 synthetic vulnerable programs, four real-world open source and industrial applications, and 19 demos. We discovered eight crashes and two previously unknown vulnerabilities, demonstrating the effectiveness of Inception as a tool to assist embedded device firmware testing.

1 Introduction

Embedded systems combine software and hardware and are dedicated to a particular purpose. They generally do not have the traditional user interfaces of desktop computers. Instead, they interact with the environment through several peripherals, which are hardware components that handle sensors, actuators, and communication protocols. The constant decrease in the cost of microcontrollers, combined with the pervasiveness of network connectivity, has led to a rapid deployment of networked embedded systems being used in many aspects of modern life and industry. These trends have greatly increased embedded systems' exposure to attacks. The consequences of a vulnerability in embedded software can be devastating. For example, the boot Read Only Memory (ROM) vulnerability used to jailbreak some iPhones cannot be patched in software, because the bootloader is hard-coded in the ROM [12]. Therefore, it is very important to thoroughly test such low-level embedded software. Unfortunately, the lack of tools, the intricacy of the interactions between embedded software and hardware, and short deadlines make this difficult.

Binary or source-based testing. The conditions under which testing is performed can vary a lot depending on the context. The tester may have access to the source code, or just the binary code, and may use the device during testing or rely on simulators. Binary-only testing is frequently performed by third parties (pen-testing, vulnerability discovery, audit), whereas source code-based testing is more commonly done by the software developers or when the project is open-source. Access to source code provides many advantages; such as knowing the high-level semantics (e.g., the type of variables) of the program. This simplifies testing significantly.

An advantage of binary-only testing is that it can be performed independently of source code availability, and is, therefore, more generic. Indeed, even when source code is available, it can be compiled and the analysis

	FIE [10]	SURROGATES [17]	Avatar [34]	Inception
Using source code	✓	✗	✗	✓
Inline assembly	✗	✓	✓	✓
Binary code	✗	✓	✓	Some
Symbolic execution	✓	✗	✓	✓
Can use real peripherals	✗	✓	✓	✓
Early bug detection	✓	*n/a*	✗	✓
Fast forwarding	*n/a*	✓	✗	✓
Fast concrete execution	✓	*n/a*	✗	✓
Testing unmodified code	✗	✓	✓	✓
Low false positives	✗	*n/a*	✓	✓
Highly automated	✗	*n/a*	✗	✓
Open-source	✓	✗	✓	✓

Table 1: Comparison of Inception with the related work.

can be performed on binary software. Unfortunately, this is inefficient, because during compilation, most code semantics are lost and this renders identification of memory safety violations and corruptions difficult. In fact, it has been shown that this effect is more severe with embedded software than with regular desktop software, due to the frequent lack of hardening of embedded software and hardware support for memory access controls such as memory management units [23]. Also program hardening (e.g., with Sanitizers [30]) is often impossible due to code space constraints and the lack of support for embedded targets.

Hand-written assembly. Unfortunately, the presence of hand-written assembly and third-party binary libraries is widespread in embedded applications. This severely limits the applicability of traditional source-based testing frameworks. There are two main reasons for the use of assembly language in embedded software development. First, although memory becomes cheaper and compiler efficiency improves, it is still often necessary to manually optimize the code (e.g., to fit in the cache, to avoid timing side-channels) and microcontrollers' memory size is still very constrained. Assembly is also necessary to directly interact with some low-level processor features (e.g., system-control or co-processor registers, supervisor calls).

Figure 1 highlights this problem on a set of sample programs from our test-suite (described in Section 4). Every sample contains at least one function with inline assembly. We further distinguish four categories of instructions, based on how they affect the system. From left to right: logical (e.g., arithmetic, logic), memory (load, store, barrier), hardware (supervisor call, co-

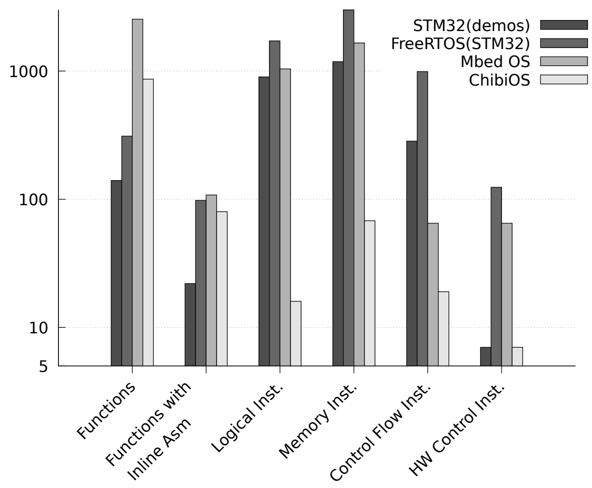


Figure 1: Presence of assembly instructions in real-world embedded software.

processor registers access), control-flow (branch and conditional).

Logical and memory instructions are easy to translate to higher-level code. However, hardware *impacting* instructions strongly interact with the processor and affect the execution and the control flow. Common source-based frameworks cannot easily handle these low-level instructions. However, they are essential to handle tasks such as context-switching between threads. As a consequence, replacing those instructions with high-level code is difficult. We found that such instructions are present in all of the samples. Other places where assembler instructions or binary code is present is in Board Support Packages (BSP) provided by chip manufacturers or in library code directly present in ROM memory.[1]

Previous work. Table 1 summarizes the limitations of firmware security analysis tools. Avatar [34] and SURROGATES [17] focus on forwarding memory accesses to the real device, but only support binary code. Avatar relies on S2E [8] and, therefore, supports symbolic execution of binary code. On the other hand, FIE [10] tests embedded software using the source code, essentially adapting the KLEE virtual machine to support specific features of the MSP430 architecture. However, FIE does not try to simulate hardware interaction: writes to a peripheral are ignored and reads return unconstrained symbolic values. Moreover, FIE does not support assembly code which is very often present in such software and is, therefore, either entirely skipped or manually replaced by equivalent C code, if possible. This requires additional manual work, makes the state explosion worse, and leads to a less accurate emulation.

Inception's approach. Inception's goal is to improve

[1]For example, the NXP MC1322x contains drivers and a Zigbee software stack in a mask ROM [24].

testing embedded software when source code is available, e.g., during development phases. We focus on the ability to perform security testing on complete systems made of real-world embedded software that contain a mix of high-level source code, hand-written assembly code, and, possibly, binary code (e.g., libraries). Unlike previous work, in Inception we preserve most of the high-level semantics from source code. We, therefore, can test software against real hardware peripherals with high performance and correct synchronization. Finally, to be broadly used, such integration tests need to be performed with a limited amount of manual work.

Contributions. In summary, in this paper we present the following contributions:

- A new methodology to automatically merge low-level LLVM bitcode, poor in semantic information and relying on the features of a target architecture, with high-level LLVM bitcode, rich in semantic information useful to detect vulnerabilities during symbolic execution
- A modified symbolic virtual machine, able to run the resulting bitcode code and to handle peripherals' memory and interrupts using different analysis strategies
- A fast debugger to connect the peripherals on the real device with the virtual machine, preserving event synchronization
- A thorough validation of the system to guarantee meaningful and reproducible results, and an evaluation of the approach on both synthetic and real-world cases
- A tool based on affordable off-the-shelf hardware components and source code that will be fully published as open-source

Paper organization. The remainder of the paper is organized as follows. Section 2 provides an overview of the approach and introduces the Inception tool. Section 3 presents the main implementation challenges and our validation methodology. Section 4 evaluates Inception on synthetic and real-world cases. Section 5 discusses limitations and future work. Section 6 reviews related work and, finally, Section 7 concludes the paper.

2 Overview of Inception

2.1 Approach and components

The main goal of Inception is to leverage the semantic information of high-level source code to detect vulnerabilities during symbolic execution, while also supporting low-level assembly code and frequent interactions with the hardware peripherals. Common symbolic execution environments usually run an architecture-independent representation of the code, which can be derived from the sources without losing semantic information. Alternatively, architecture-dependent binary code can be lifted to an intermediate representation that can be at least partially executed into a symbolic virtual machine, but that has lost the source code semantic information. These two cases differ greatly (e.g., in their memory model) and cannot easily coexist.

Inception solves the problem of coexistence by creating a consistent unified representation. In particular, Inception is composed of three parts. First, the **Inception Translator**, which generates unified LLVM-IR using a lift-and-merge process to integrate the assembly and binary parts of the program into the intermediate representation coming from the high-level sources. This process also takes into account the low-level hardware mechanisms of the ARMv7-M architecture. Second, the **Inception Symbolic Virtual Machine**, which is able to execute this mixed-level LLVM-IR, and to handle interrupts and memory-mapped peripherals with different strategies, to adapt to different use cases. It can also generate interrupts on demand and model reads from peripherals' memory as unconstrained symbolic values. This VM is based on KLEE, a well-known open-source symbolic execution virtual machine which runs LLVM-IR bitcode. Third, the **Inception Debugger**, which is a custom fast debugger, built around a USB3 bus adapter and an FPGA. It provides high-speed access to the peripherals and could be easily extended for multiple targets.

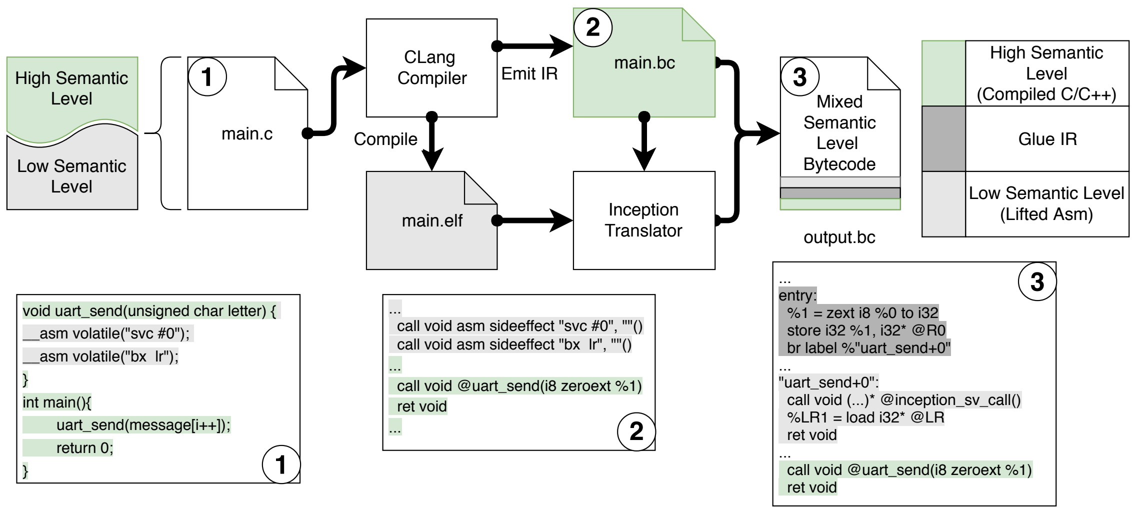
In the following we give an overview of our lift-and-merge approach, of how KLEE performs security checks, and on how we extended it to support interrupts and peripheral devices.

2.2 Lift-and-merge process

Figure 2 shows the main stages of our bitcode merging approach and how source code with inline assembly ① is transformed into a consistent bitcode ③ that can be executed by Inception VM. The example code contains the excerpt of a function written in assembly that requests a system call with `r0` holding a data byte.[2]

The rest of the code is composed of a main function, which calls the first assembly function, and the message to be sent. Using the appropriate LLVM front end (CLang for C/C++), source code ① is translated into LLVM-IR bitcode. The resulting bitcode ② shows that only C/C++

[2]Figure 10 in the appendix shows the complete example, including the system call handler (in assembler) which sends the data byte over a UART by writing into the data register of the UART peripheral.



```
void uart_send(unsigned char letter) {
__asm volatile("svc #0");
__asm volatile("bx  lr");
}
int main(){
        uart_send(message[i++]);
        return 0;
}
```

```
...
 call void asm sideeffect "svc #0", ""()
 call void asm sideeffect "bx  lr", ""()
...
 call void @uart_send(i8 zeroext %1)
 ret void
...
```

```
...
entry:
  %1 = zext i8 %0 to i32
  store i32 %1, i32* @R0
  br label %"uart_send+0"
...
"uart_send+0":
  call void (...)* @inception_sv_call()
  %LR1 = load i32* @LR
  ret void
...
  call void @uart_send(i8 zeroext %1)
  ret void
```

Figure 2: Overview of *Inception Translator*: merging high-level and low-level semantic code to produce mixed semantic bitcode. Excerpt of the translation of a program which includes mixed source and assembly.

source code has been really translated into LLVM-IR. Indeed, the original purpose of LLVM-IR bitcode is to enable advanced optimizations before code lowering to the target architecture, whereas assembly is already at a low semantic level that cannot be represented or optimized by the LLVM compiler.

To solve this problem, we introduce a novel lift-and-merge approach, which we implement in *Inception-Translator*. This translator takes as input the ELF binary and the LLVM-IR bitcode generated by CLang. It generates a consistent LLVM-IR bitcode where assembly instructions have been abstracted to an LLVM-IR form. This step is done by a static lifter, which replaces each assembly instruction by a sequence of LLVM-IR instructions. We call the resulting bitcode a Mixed Semantic Level bitcode (mixed-IR), shown in (3), which contains:

High Semantic Level IR (high-IR) obtained from C/C++ source code. This is mainly the same code emitted by CLang, which has been augmented with external global variables that are defined in assembly source files. We reallocate these global variables in the IR.

Low Semantic Level IR (low-IR) deriving from assembly source code. This part is automatically generated by our static lifter. It contains the translation of assembly instructions and some architecture-dependent elements that are necessary for execution. First, the CPU and co-processors' registers are modeled as global variables. Second, specific functions model the seamless hardware mechanisms that are normally handled by the CPU. For example, when entering into an Interrupt Service Routine (ISR), the processor transparently updates the Stack Pointer and it stacks a subset of CPU registers. When the ISR returns, the context is automatically restored, so that the code which was suspended by the interrupt can resume.

The Glue IR that acts as a glue to enable switching between the high-level semantics and the low-level semantics domains. This IR bitcode is generated by a specific Application Binary Interface (ABI) adapter, able to promote or demote the abstraction level. Indeed, communication and switching between layers mainly happens at the interface between functions, that is, when a high-level function calls a low-level one or the opposite.

2.3 Inception Symbolic Virtual Machine

The bitcode resulting from the lift-and-merge process is almost executable, but it still requires some extra support in the virtual machine. The main challenge is that high-IR accesses only typed variables and does not model memory addresses or pointers. On the other hand, the IR generated from assembly instructions has lost all information about types and variables, and only accesses pointers and non-typed data. Another challenge is handling memory-mapped memory, which is used but not allocated by the code, and interrupts and context switches, which are not modeled in KLEE.

To address these problems, we have extended KLEE with a *Memory Manager* and an *Interrupt Manager*. During (symbolic) execution the original *Memory Monitor* of KLEE performs advanced security checks on memory accesses. When a violation is detected, the constraint solver generates a test case that can be replayed.

The Memory Manager leverages the ELF binary and the mixed-IR to build a unified memory layout where both semantic domains can access memory. Specific data regions are allocated in order to run low-IR code, such as

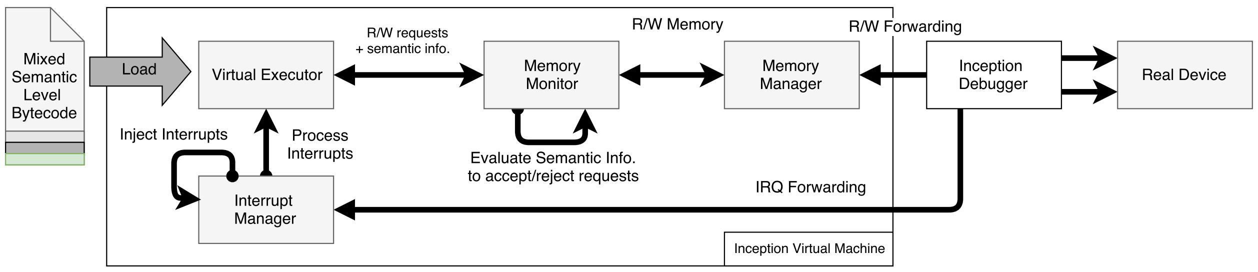


Figure 3: *Inception Symbolic Virtual Machine*, overview of the testing environment.

pointers contained in the code section, and some memory sections (stack, heap, BSS). Each memory address is configurable to mimic the normal firmware's environment. For example, a memory-mapped location could be redirected to the real peripheral, to prune the symbolic exploration and to use realistic values. Alternatively, it could be allocated on the virtual machine and marked as symbolic to model inputs from untrusted peripherals. Inception also supports Direct Memory Access (DMA) peripherals, provided that each DMA buffer is flagged as redirected to the real device memory. Similarly to the other redirected locations, DMA buffers cannot hold symbolic values.

The Interrupt Manager gives KLEE the ability to handle interrupt events, by interrupting the execution and calling the corresponding interrupt handlers. Interrupt's addresses are resolved using the interrupt vector table. Interrupt events are either collected on the real hardware, or generated by the user when desired (by calling a special handler function). In the first case, the virtual machine and the real device are properly synchronized to avoid any inconsistency. We further extended KLEE to execute handlers that switch the context between threads in multithreaded applications.

Memory Monitor and security checks. All security analyses mainly rely on the Memory Monitor of KLEE, which is able to perform security check for each access, based on the semantic information associated to it. The monitor observes the semantic information of the requests (requested type) and the semantic information of the accessed data (accessed type). When enough information is available, the monitor is able to detect memory access violations, e.g., out-of-bounds accesses, use-after-free, or use-after-return. Requests coming from high-IR, and accessing memory elements defined in high-IR, have enough information to detect most violations. On the contrary, requests that come from low-IR tend to have less information and a lower detection rate. However, thanks to the information coming from the high-IR, it is still possible to detect more problems than with binaries only.

3 Implementation and validation

3.1 Lift-and-merge process

In order to be able to glue assembly and binaries with source code into a unified LLVM-IR representation (mixed-IR), we apply two distinct processes.

The lifting process takes machine code (compiled assembly or binaries) and produces an equivalent intermediate representation (low-IR). This representation uses only low-level features of the LLVM-IR language and it mimics the original architecture (ARMv7-M), which contains some hardware semantics of the Cortex-M3 processor, such as the behavior of instructions with side effects. It is, therefore, (almost) self-contained, and a large part of it can be executed on any virtual machine able to interpret LLVM-IR. As explained in the following parts, we introduce some features to KLEE to make this code fully executable, in particular when dealing with context switches. Our lifter is based on three main components. First, a static recursive disassembler that finds all the instructions to translate and stores them into an internal graph representation. Second, a simple decompiler that reconstructs the control flow, including for indirect branches and complex hardware mechanisms (e.g., returns from interrupts and context switches). Finally, the lifter statically transforms a given machine instruction into a semantically equivalent sequence of LLVM-IR instructions. One important advantage of the static approach is that it enables further processing with the sources to produce mixed-IR. Moreover, it has a lower run-time overhead compared to dynamic lifters that lift instructions during execution. Implementing all these components in a correct and reliable way requires significant engineering work[3], for which we omit most of the uninteresting details. In the next section we will describe some interesting aspects of the lifter.

[3]We first used Fracture [18], a framework for lifting binaries to LLVM-IR. However, we eventually only reused a minor part of Fracture code. Indeed, Fracture's approach does not scale to all instructions, especially those interacting with hardware, and does not address the merging problem. Fracture was also designed for static analysis which did not need complete translation and is currently not maintained.

The merging process takes the (almost) self-contained low-IR and the high-IR compiled from C/C++, to glue them together (with some glue-IR). This is the most challenging part, as they have different levels of semantic information and different views of memory. The first step is, therefore, to create a unified memory layout between the two IR-levels in the KLEE virtual machine. In addition to this, peripheral device addresses are made accessible in KLEE. The second step consists of identifying the best interface between the two representations and the mechanisms to exchange data at this boundary. We chose to use the Application Binary Interface (ABI) that regulates the communication between functions in a uniform way.[4] Our merger is able to generate glue-IR code that lets high-IR functions communicate with low-IR functions and vice-versa.

3.2 Unified Memory Layout

We now explain how we leverage both the lift-and-merge process and KLEE to create a unified memory layout. This memory layout is central for the low-IR and high-IR to coexist and communicate.

Processor registers are represented by global variables for different reasons. First, the LLVM-IR is a Single Static Assignment (SSA) language, in which each instruction stores its result in a uniquely assigned register. Secondly, LLVM supports an unlimited number of registers, which are assigned only once and are not globally accessible. Therefore, LLVM registers cannot be used to represent CPU registers, which are limited, assigned many times, and globally accessible by instructions.

The heap. Inception supports two dynamic memory allocation mechanisms. The first one is the native allocation function from the application (which can be written in assembly or C language). In this case, allocated variables lose semantic information and are encased in the heap memory region. This method is interesting for testing native allocation systems. However, it decreases the precision of corruption detection, because the heap memory is a container for indistinguishable contiguous variables, making it difficult to detect even simple out-of-bounds accesses. The second approach consists of replacing the native allocation functions by KLEE's own allocator. KLEE allocator was specifically designed to detect memory safety violations. In particular, KLEE isolates each allocated variable with a fixed-memory region (the *red zone*). Even though this mechanism does not detect all violations, any access to this zone will be detected as a memory corruption. Another advantage of KLEE allocation is that it can detect memory management errors such as invalid free of local or global variables.

The normal KLEE stack is used when high-IR code is running. Each function has its own function frame object, which contains metadata about the execution. This includes information about the caller, the SSA registers values (which hold temporary local variables), and the local variables (which are allocated using the normal KLEE mechanism). A separate **stack** is used by the low-IR code. This stack is modeled as a global array of integers, allocated by the memory manager at the same address and size than the .stack section of the symbol table. Variables in this stack are not typed. However, the ABI adapter mechanism presented in the next section allows different IR levels to access variables on both stacks.

The Data region contains mixed semantic-level variables. Indeed, when the high-IR allocates data, the resulting memory object is typed and allocated at the same address as indicated by the symbol table, to keep the compatibility with assembly code. On the other hand, data can be defined by the assembly code and accessed by high-IR. In this case, we use the semantic information present in the external declaration of the high-IR to allocate a typed object. The third possible case is data allocated by assembly code, but never accessed by high-level code. In this case no semantic information is present, and allocation depends on the information from the ELF symbol table.

3.3 Application Binary Interface adapter

Low-IR functions follow the standard Arm Application Binary Interface (ABI) [2], whereas high-IR functions follow the LLVM convention. Therefore, whenever the *Static Binary Translator* finds a call or return that crosses the IR levels, it invokes the *ABI adapter* to generate some glue-IR that adapts parameters and return values.

When a high-IR function calls a low-IR function, the high-IR arguments (typed objects) must be lowered to the architecture-dependent memory (stack/CPU registers). In the opposite case, stack and CPU registers must be promoted to high-IR arguments. Similar considerations apply to return values. This process is similar to serializing and deserializing the LLVM typed objects, to store them as words in the LLVM variables that represent the CPU registers and the stack, where they are used by low-IR. Note that during serialization the types are lost, but deserialization is still possible thanks to the high-level information present in the source code. For example, consider an assembly function that passes a `struct` by value to a C function. Knowing the size and address of the destination, the adapter generates the glue-IR that copies CPU registers and stack words from the low-IR

[4]Another option would be to set the interface at the native instruction level. An advantage would be to preserve most of the code translated from the high-IR in a function that includes only one inline assembler directive. However, the interfacing would depend on the compiler version and would be less robust.

to the high-IR destination. Another example is an assembly function that returns a pointer. In low-IR, the pointer is stored as a simple integer word in the `r0` register. Since the adapter knows that the expected return type is a pointer, it can write the glue-IR that performs the cast to it. All main C types are supported. There are four possible connections between low-IR and high-IR (code examples available in the appendix):

1. **High-IR to low-IR parameters passing**. A glue-IR prologue takes the input arguments from the KLEE stack (where the high-IR caller stored them) and brings them to the CPU registers and/or low-IR stack (where the low-IR callee expects them).

2. **Low-IR to high-IR return value**. A glue-IR epilogue takes the return value (stored in `r0` by the low-IR callee) and promotes it to a typed object in KLEE stack (used by the high-IR caller).

3. **Low-IR to high-IR parameter passing**. Before calling the high-IR function, some glue-IR takes the input arguments from the CPU registers or the low-IR stack (where the low-IR caller stored them) and promotes them to typed objects on the KLEE stack (used by the high-IR callee).

4. **High-IR to low-IR return value**. Just after the high-IR callee returns, some glue-IR moves its return value from the KLEE stack to `r0`.

3.4 Noteworthy control-flow cases

We focus on the explanation of noteworthy control-flow instructions and hardware mechanisms to show their impact for the security checks. We omit the details for the other instructions.[5]

Control-flow instructions. The main challenge when dealing with control flow consists in finding a good mapping between high-level control flow operators present in LLVM-IR (e.g., `call`, `if/else`) and low-level ARMv7-M instructions, which are at a lower abstraction layer (they directly modify the program counter, and sometimes rely on implicit hardware features).

We translate to an LLVM call instruction any Arm instruction that saves the program counter before changing its value (i.e., direct and indirect branch-and-link instructions) to an LLVM call instruction. In order to support indirect calls, we leverage an optimization technique called *indirect call promotion* [1, 20, 7, 31]. This technique consists in transforming each indirect call into direct conditional branches and direct calls. Indirect call promotion has been introduced to improve the performance of branch prediction [1]. Conditional branches compare the target address of the indirect call with the entry point of each possible function in the program. If the condition is true, this function is called directly. This is equivalent to enforcing a weak control flow integrity policy, and akin to what KLEE already does for C/C++ function pointers. It would be possible to enforce stricter control flow integrity checks by retrieving the control flow graph with a static analysis or a compiler pass.

We translate all instructions that restore the previous program counter, for example `bx lr` and `pop pc`, to return instructions. These returns still work as intended even if the return address is corrupted. However, we do not rely on side effects (return to a corrupted address) to detect corruption. We rather detect the corruptions by relying on the memory checks, e.g., to detect buffer overflows.

We implement all other direct (conditional) branches and `it-blocks`[6] with simple direct branches available in LLVM-IR.

Interrupts and multithreading. The control flow of the program is also modified by interrupts, which asynchronously block the normal execution and call-defined handler functions. Interrupts are used very frequently in embedded programs to synchronize the peripherals with the embedded software in an event-driven fashion, or to implement multithreading.

Inception VM can receive interrupts from the real device (when real peripherals are used and generate interrupts) or generated by the user using helper functions (e.g., to stress specific functions in a deterministic way). We extended KLEE so that the main execution loop checks for the presence of interrupts to serve. In this case, KLEE executes an LLVM-IR helper function that accesses the interrupt vector table in the firmware memory to resolve the address of the interrupt handler to call, based on its identifier (ID). This dynamic resolution is necessary only if the firmware overwrites the vector table. If the vector is fixed, a slight speedup in execution can be obtained by storing the vector in a configuration file, loaded by KLEE at startup.

Before giving control to an interrupt handler, and when returning from it, a Cortex-M3 processor performs several seamless operations (e.g., stacking and unstacking the context, managing two stack modes). In Inception, a special glue-IR helper function generated by our lift-and-merge process performs these steps.

To implement multithreading, operating systems such as FreeRTOS use the interrupt and stack management features offered by the Cortex-M3. In summary, the operating system, which has its own stack, manages a separate stack for each thread. Context switching is pos-

[5]The lifting of these instructions is similar to re-implementing a Cortex-M3 in LLVM-IR based on the ARMv7-M reference manual.

[6]In ARMv7-M an "`it-block`" is a group of up to four instructions executed only if condition of a preceding `it` instruction is true.

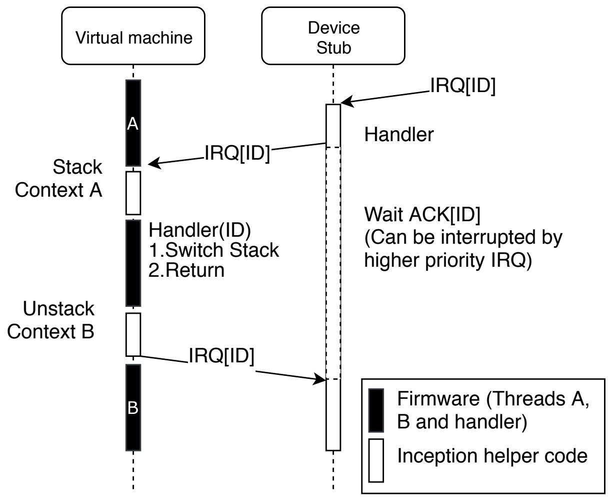


Figure 4: Context switch due to an IRQ.

sible because when a thread is interrupted, its context is saved to its stack, and the context of the resuming thread, including the program counter, is pulled from another stack. The switch is done in part by the processor and in part by the operating system. Inception fully supports this process, since all the required features are self-contained in the mixed-IR. *Inception VM* extends KLEE's call stack management, to be able to handle one call stack for each thread. Briefly, whenever a new thread is spawned, a new call stack structure is generated and assigned to it.

Synchronization with the real device. To collect interrupts on the real device, we insert a stub on the device that registers one handler for each possible interrupt. When an interrupt is fired, the handler is called and notifies KLEE thanks to the forwarding system. The main challenge of this architecture is to keep the virtual machine and the device synchronized, without inconsistencies and race conditions, even in presence of multiple priorities. This needs to be done carefully and uses several mechanisms. In particular, the interrupt handler on the device should not return until the corresponding KLEE handler terminates. This is necessary, for example, to mask interrupts with the same or lower priority until the handler ends, as it happens in the real device, and to avoid the flooding of new interrupts.

A complete example. Figure 4 shows an example of context switch triggered by an interrupt generated on the device. On the right we see how the identifier of the interrupt is used both to notify KLEE at the beginning and to acknowledge the stub at the end. The acknowledgement is per-identifier, so that the stub can be interrupted by higher priority interrupts. On the left, we can observe the switch between threads enabled by the seamless context stacking and unstacking.

In summary, *Inception Debugger* fully handles interrupt synchronization with the host virtual machine, while previous work had only limited interrupt support [34].

3.5 Forwarding mechanism with *Inception Debugger*

In the previous parts we described how we integrated peripheral devices and interrupts in the virtual machine. We now focus on the lower layers of the communication mechanism between the host and the real device.

In order to read and write the device memory, we directly connect to the system bus through the AHB-AP, which can be accessed with the JTAG protocol.[7] The AHB-AP port is available in Arm Cortex-based devices and allows a direct access to the peripherals. Inspired by SURROGATES [17], we designed a custom device based on a Xilinx ZedBoard FPGA [11], to efficiently translate high-level read/write commands into low-level JTAG signals.[8] The FPGA is connected through a custom parallel port to a Cypress FX3 device [29] which provides an USB3.0 interface. Unlike USB2 where devices are slaves, USB3 is a point-to-point protocol and, therefore, has a very low latency. With this setup we handle the burden of the low-level and inefficient JTAG protocol in hardware close to the device, while we transmit high-level commands over a low-latency high-bandwidth bus to/from the host. Our debugger is able to communicate with the stub running on the device and handle interrupts using a dedicated asynchronous line and shared memory locations.

In summary, we provide a clean slate design for an efficient, cheap , and open-source solution, which can be used to experiment and replicate research that requires customizable debuggers (e.g., [25]).

3.6 Validation

We carefully validated Inception to obtain a reliable tool.

Regression Tests. We created a framework for automated regression testing of the code. Around 53200 tests are performed at several levels of abstraction, from unit tests up to tests involving all components. Results are compared to a *Golden model* (i.e., a known and trusted reference). For example, we compared single instructions against the real Cortex-M3 processor, assembly functions against the C code from which they originate or alternative implementations, and complete applications

[7] An alternative would be a port using the faster SWD protocol, but this technology is less widespread than JTAG.

[8] SURROGATES [17] was never open sourced but the authors shared their implementation. However, due to lack of hardware availability and other problems we eventually re-designed the debugger from scratch.

against their behavior on the native hardware. We stress symbolic execution on known control flow cases, and bug detection on known vulnerabilities.

Arm Cortex-M3 lifter. The correctness of the lifter is particularly important to obtain correct execution. Our framework generates all possible supported instructions, starting from a description of the instruction set. Then, for each type, it creates several tests with random initialization of registers and stack. Finally, in executes them both on the device and in Inception, and it compares the final state of registers and stack. Table 4 in the appendix summarizes all the tests we preformed.

4 Evaluation and comparison

After validation, we evaluated Inception over a set of interesting samples, which we explain in this section. We first focus on the effects of semantic information on vulnerability detection and on the speed performance of the tool. Then, we show analyses on more complex examples including, for example, assembly code for multithreading and statically linked libraries. Finally, we explain how Inception found corruptions in three industrial applications under development, including a boot loader. Evaluating and comparing tools for embedded software analysis is hard because of the lack of an established benchmark suite. This is rendered harder due to the large number of different hardware platforms. While some of the examples we use below are proprietary, we also built a large set of validation and evaluation examples, sometimes based on existing open-source code. Those examples will be made available together with Inception and may provide a basis for such a benchmark.

4.1 Vulnerability detection

Detection rate at different semantic levels. We evaluate how vulnerability detection is affected by the semantic level of high-IR and low-IR and their interaction. In particular, we explore if KLEE can detect memory corruptions on a vulnerable path, depending on how variables are allocated and accessed by different types of IR. Our analysis samples are based on the *Klocwork Test Suite for C/C++*[9], which includes out-of-bound, overflow, and wrong dynamic memory management errors. We initially compile them to high-IR (and binary). We then selectively force the decompilation from binary to low-IR of some functions, obtaining 40 different interaction cases. Table 2 summarizes the different combinations of allocation and access of memory objects at different semantic levels, and the consequent detection result, which we comment in the following.

[9] `https://samate.nist.gov/SRD/view.php?tsID=106`

First, detection works only for those memory objects allocated in high-IR for which we have semantic information. However, the memory accesses can come from both high-IR and low-IR or be related to the return value of low-IR functions. For example, a C function allocates a buffer that is then improperly used by an assembly function. If the called function overflows the buffer, it will access an unallocated memory space of the high-IR domain where memory objects have a defined size, type and which are separated from each other by a red zone. The semantic information of high-IR memory objects greatly improves the detection of vulnerabilities even if it occurs in low-IR code. However, if the buffer is allocated by a low-IR code (assembly or binary code), the lack of semantic information about the variable prevents the detection of the overflow. The same mechanism is applied to local (static) allocation and global allocation.

Second, when using KLEE *dynamic allocation functions, all vulnerabilities can be detected in both high-IR and low-IR,* whereas if we use some implementation in the code of the application, the detection rate drops to almost zero for both high-IR and low-IR. However, in this case we can test the code itself of the allocation functions, either in high-IR or low-IR depending on the case.

In summary, in 40 synthetic tests, 70% of the inserted vulnerabilities were found and no false vulnerabilities were reported.

Comparison with binary-only approaches. When testing embedded binary code, it is hard to catch memory corruptions because of the lack of semantic information, code hardening, and operating system protections. For example, [23] highlights the problem when fuzzing a STM32 board, and it uses several heuristics to catch corruptions. To compare this approach with Inception, we analyze the same firmware (EXPAT XML parser with artificial vulnerabilities). Each vulnerabilty (stack/heap-based buffer overflow, null pointer dereference, and double free) has its own independent trigger condition. We start with the source code compiled to high-IR, but we also generate cases with low-IR by forcing the decompilation of vulnerable functions. To use Inception, we mark the input as symbolic and run the samples with a timeout of 90 s. Results are visible in Figure 5. Our approach successfully uses all the semantic information available, keeping a good detection rate even in presence of some low-IR code. We could integrate the heuristics from [23] to improve results even further. One of the vulnerabilities could be detected, but it is not triggered because of state explosion (47k states) and the constraint solver (using 67.5% of the time), which are problems inherent to symbolic execution and common to KLEE.

Table 2: Overview of memory checks between LLVM code at different IR semantic level.

		Allocation					
		C with KLEE Allocator		C Native Allocator		ASM or Binary	
		Accessed from					
		C	ASM	C	ASM	C	ASM
Dynamic Allocation	Check Types	✓	✓	✗	✗	✗	✗
	Red Zone	✓	✓	✗	✗	✗	✗
	Heap Consistency Checks	✓	✓	✗	✗	✗	✗
Stack Allocation	Check Types	-	-	✓	✓	✗	✗
	Red Zone	-	-	✓	✓	✗	✗
.Data or .BSS Allocation	Check Types	-	-	✓	✓	✗	✗
	Red Zone	-	-	✓	✓	✗	✗
Not Allocated Memory	KLEE Detection	-	-	✓	✓	✓	✓

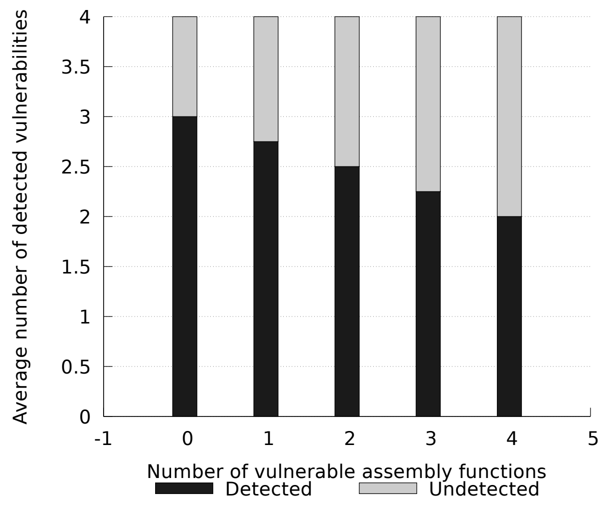


Figure 5: Evolution of corruption detection vs. number of assembly functions in the EXPAT XML parser (4 vulnerabilities [23], symbolic inputs, and a timeout of 90 s).

4.2 Timing overhead

Overhead of the executor. We evaluate the execution speed of the virtual machine using the DHRYSTONE[10] v2.1 benchmark, compiled without any optimization in LLVM-IR. Inception has 38% of slowdown overhead compared to KLEE, but if we disable the multithreading support the overhead becomes insignificant. Inception is 17 times slower than the real hardware[11]. This is mostly due to execution in the KLEE virtual machine.

Overhead of low-IR (advantage of high-IR). One of the advantages of our source-based approach is that we maximize the use of high-IR, which is more compact and faster than low-IR. To provide a rough example, we force 3 functions out of 12 in DHRYSTONE V2.1 to be translated from binary, which is a realistic proportion. This adds 343 more IR lines to the initial 1636, reducing the speed by around 43%. Low-IR does not seem to affect the time spent in the constraint solver. For example, we run `bubble sort` and `insertion sort`, with a symbolic array of 10 integers and a timeout of 90 s. Both the high-IR and the low-IR versions spend about 90% of the time in the constraint solver.

Overhead of forwarding. *Inception Debugger* has a read/write performance comparable to the fastest similar debugger (SURROGATES [17]). Using JTAG at 4 MHz, reads are 20% slower and writes are 37% faster in Inception (Table 6). It seems that in our implementation the bottleneck comes from the USB software stack, rather than from JTAG, which can easily run faster, or from the USB protocol, which has itself a very low latency. Indeed, the GNU/Linux userspace library (`libusb-0.1-4`) performs system calls and DMA requests for each I/O operation, introducing a significant latency. Using bulk transfers of 340 reads is five times faster, since the latency for a USB operation appears only once. Unfortunately, code execution requires single memory accesses, but bulk tranfers could be used when dealing with DMA forwarding to reduce latency, SURROGATES uses a custom driver that exposes FPGA registers through MMIO over PCI-Express. Though the exact same approach is not possible, using a custom driver may improve Inception performance.

Benchmark of some real applications. We evaluate the overall performance (software stack and forwarding) of three popular protocols: ICMP, HTTP, and UART. For the first two we use the `Web`[12] example for the `LPC1850`

[10]DHRYSTONE is a synthetic computing benchmark program, available at `http://www.netlib.org/benchmark/dhry-c`.

[11]Value reported by the manufacturer for a STM32 with Cortex-M3.

[12]It is part of the `lpc1800-demos` pack available at `https://diolan.com/media/wysiwyg/downloads/lpc1800-demos.zip`

Type	Total	Detected	Rate
Division by Zero	88	88	100%
Null Pointer Dereference	131	131	100%
Use After Free	62	62	100%
Free Memory Not on Heap	1.131	1.131	100%
Heap-Based Buffer Overflow	38	38	100%
Integer Overflow	112	0	0%
Total	1.562	1.450	92%

Table 3: Corruption detection of real-world security flaws based on FreeRTOS and the Juliet 1.3 test suites.

board. We use the Ethernet interface of the real device, forwarding memory accesses and interrupts. In particular, we identity the DMA buffers and configure Inception to keep them on the memory of the real device. For the UART, we use the driver of the `STM32` board, again using the real peripheral. For all protocols we use simple clients (`ping`, `wget`, and `minicom`) on a laptop, and we repeat measurements for 100 runs. Results are shown in Figure 7. There are two reasons why ICMP and HTTP are slower than UART. First, they have a more complex software stack. Second, they require forwarding of many interrupts and of large DMA buffers.

4.3 Analysis on real-world code

We evaluate the capabilities of the Inception system on two publicly available real-world programs. These two samples cover the different scenarios in which Inception can be applied.

FreeRTOS is a market-leading real-time operating system supporting 33 different architectures.[13] It provides a microkernel with a small memory footprint and thread support. For this, it uses small assembly routines that strongly interact with the features of the target processor and it is, therefore, a good test case for Inception. We show that Inception can execute low-level functions that deal with multithreading before reaching vulnerable areas. We experiment with the injection of vulnerabilities in one thread, symbolic execution with producers and consumers, and corruption of the context of a thread.

We take the injected vulnerabilities from the NSA Juliet Test Suite 1.3 for C/C++, which collects known security flaws for Windows/Linux programs.[14] We selected tests related to divide by zero, null pointer dereference, free memory not on heap, use after free, integer overflow, heap-based buffer overflow. We skip tests that cannot run on our target STM32L152RE (e.g., those that require a file system or a network interface) and those that the LLVM 3.6 bitcode linker cannot handle (poor support of the C++ name mangling feature) for a total of 10384 and 1214 deletions, respectively. Furthermore, we update namespace names to comply with CLang 3.6. We obtain 1562 tests which we embed in FreeRTOS threads.

To trigger the vulnerabilities, Inception has to first execute low-level code containing assembly, and in some cases also to flag as symbolic the output of a software or hardware random generator. The interrupts required for context switches and timers can be either collected on the real device or simulated (with the appropriate generation functions). We chose the second option to be able to run many tests quickly. We set a timeout of 300 s and we observed that we can reach these regions without manual effort or modification to the multithreaded code (Table 3). The detection rate is 100% for divisions by zero, null pointer dereference, use after free, free of non-heap allocated memory, and heap buffer overflow vulnerabilities. Integer overflows are not detected at all in KLEE (version 1.3). However, we note that in general it may be possible to detect a consequence of the overflow later.

We also wrote a simple multithreading library that uses the same hardware features as FreeRTOS. On top of it, we created a simple example with three threads, where two consumers use the data put in a circular buffer by a producer. This simulates, for example, an application that processes sensor data. Depending on a symbolic value, threads execute in different order with different data. Inception can easily find a condition that triggers an overflow in the circular buffer. We also simulate the presence of a vulnerable code that corrupts the context of a thread, in particular its program counter on the stack. In this case, when the corrupted thread resumes, Inception detects that the program counter is invalid (not part of a thread that was correctly started before). Note that there may be false positives (if such behavior was intentional) or negatives (if the corrupted address is still valid).

libopencm3 is an open-source library that provides drivers for many Cortex-M devices.[15] We test some examples in which the library is a statically linked binary. It is very similar for *Inception Translator* to lift and merge a function in a statically linked library or from a function that contains inline assembly. For example, we write a sample that uses the CRC peripheral to compute the Code Redundancy Check (CRC) on a buffer. The CRC peripheral computes one word at a time, so the driver iterates over the buffer locations. Besides this, the application calls other `libopencm3` functions to initialize the STM32 device and to configure and blink LEDs. Though the driver and the other functions are translated from the binary, the buffer is part of the application code written in C; therefore, we have semantic information on its type and size. Similarly, Inception knows the memory lay-

[13] `https://www.freertos.org/`

[14] `https://samate.nist.gov/SRD/around.php#juliet_documents`

[15] `https://github.com/libopencm3/libopencm3`

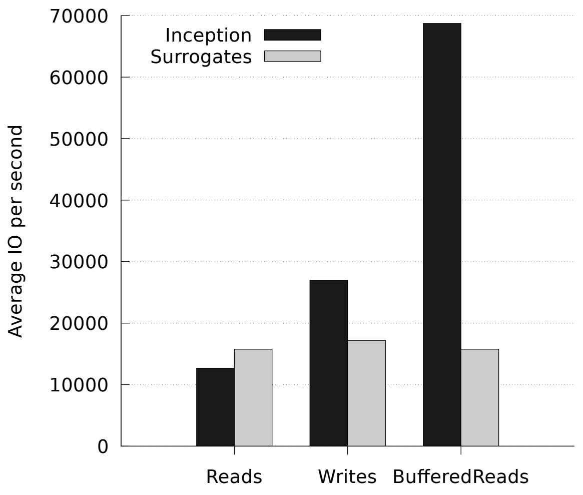


Figure 6: Average time to complete 1×10^6 read or write requests for SURROGATES and Inception (4 MHz JTAG). (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

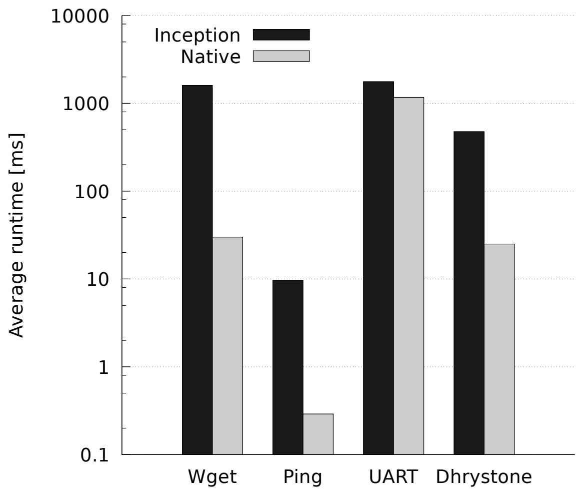


Figure 7: Performance comparison between native execution and Inception. (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

out and the location of the other variables. If the low-IR driver is called with an incorrect length parameter, this leads to an out-of-bound access which is detected by Inception. Similarly, if the buffer is dynamically allocated and erroneously freed, Inception detects a use after free. The semantic information used for detection would not have been exploited by a binary-only tool.

4.4 Usage during product development

Commercial bootloader. Bootloaders are good targets for Inception, since they contain low-level code and they often parse untrusted inputs. Moreover, they are hard to test when the real hardware is not available yet and tests on prototypes may be not accurate. To show the potential of Inception in these conditions, we analyzed a bootloader under development, and we found a problem that would have been difficult to detect on FPGA-based prototypes.

Our target is a secure bootloader with several options, stored in a One Time Programmable (OTP) memory. When it executes, the bootloader holds in SRAM a structure containing some information about the application (e.g., start address, stack address). This structure is pointed by `p_header` in the pseudo-code that follows:

```
1  void start(){
2    switch(boot_modes) {
3      case NO_SECURE_BOOT:
4        context.p_header->start_addr =
             FLASH_MEM_BASE;
5        context.stack = SRAM_STACK;
6        jump_to_application();
7      break;
8      case SECURE_BOOT:
9        do_secure_boot();
10     break;
11     default:
12       error();
13   }
14 }
```

To prepare the analysis, we configured Inception with the memory layout of peripherals. We also flagged the OTP memory as symbolic, to explore all possible paths deriving from different boot options. Despite the lack of hardware, Inception did not require any change to the source code. During symbolic execution, Inception detected a corruption (write to an invalid address) at line 4, and the solver gave us a test case to reach this condition. We manually inspected the code and confirmed that the `p_header` pointer is not initialized.

In summary, the bootloader writes a value to an address held in a non-initialized SRAM location. If the invalid write does not trigger other errors, the bootloader can still execute and successfully load the application at `start_address`, making this problem hard to detect. In particular, it does not crash on the FPGA prototype, because `p_header` is null (SRAM zeroed at reset), which is mapped to writable memory. A write to 0 would instead produce a memfault on the real device, as 0 would be mapped to a read-only memory. Detecting the bug later in the development process, like on silicon, would be expensive.

From a security perspective, an attacker may at least partially control the value of `p_header`. For example, we could imagine a scenario in which certain options lead to writing this location, and a fast reboot preserves it (SRAM is not initialized). Besides changing the destination before the write, an attacker could change it after, so that the bootloader would dereference a wrong `start_address` at which to load the application.

Chip SDK. We tested a Software Development Kit (SDK) for a commercial chip, at a stage when a prototype of the hardware was not even available yet. Therefore, we configured reads to peripherals to return unconstrained symbolic values. Inception found a test case in which a bit-wise shift depended on an untrusted value (overshift), which we confirmed by manual inspection. In this case, the error leads to the wrong configuration of a peripheral and unexpected behavior. More generally, overshifts could lead to overflows or out-of-bound accesses. Early detection is useful to avoid expensive fixes later.

Commercial payment terminal To show the potential of Inception when hardware is available, we tested a payment terminal under development, using the FPGA prototype to redirect most peripherals and their interrupts. The application communicates with an external smart card through a card reader, which we mark symbolic since it is not trusted. This mix of concrete and symbolic peripherals effectively explores the code, avoiding state explosion. Inception found eight potential vulnerabilities (out-of-bound accesses), that have been reported to developers and still have to be confirmed.

5 Discussion

In the following we discuss the advantages and limitations of Inception.

Application vs. (software/hardware) environment. The key to using symbolic execution in realistic settings is to limit the expensive symbolic exploration to a small critical code region, treating the (software/hardware) environment separately. S2E investigates how different strategies to cross this partition affect the analysis. Inception offers several options. Dynamic allocation can be either part of the environment (host functions with concrete or concretized inputs), or part of the code under test (where symbolic values can propagate). The former reduces the symbolic space at the price of completeness, whereas the second one preserves completeness at the price of higher complexity. A peripheral can be treated as a stateless untrusted function that ignores inputs and returns unconstrained symbolic values. This leads to the exploration of all possible paths, also those that would not be globally feasible with the real peripherals (making false positives possible). Though useful for drivers when the hardware is not yet available, this option does not scale because of state explosion. Alternatively, Inception can use the real peripherals with concrete values, reducing the problem. Globally unfeasible paths are reduced too, but they could still appear if the states of peripheral and code become inconsistent (e.g., if symbolic execution switches state during the access pattern to a stateful peripheral). However, symbolic exploration visits the higher-level logic of the application rather than the drivers, making the problem less common. A more thorough study is left as future work. A complete testing of a firmware program would require considering interrupts at any single instruction, which in practice is not feasible. Previous work [26] reduces the frequency of timer-based interrupts by executing them only when the firmware goes in low-power interrupt-enabled mode. However, this solution can miss issues that may occur when interrupts are processed during the firmware execution. Inception enables users to generate interrupts on demand that are useful to obtain deterministic sequences or to stress the code, but it is neither complete nor guaranteed to try cases that are actually possible. Collecting the interrupts from the real hardware covers realistic cases without additional complexity, but suffers from possible inconsistencies as explained for peripherals. We plan to analyze enable/trigger patterns to detect which symbolic states must serve an interrupt when it arrives.

Semantic gap. Inception increases the overall vulnerability detection rate for applications containing assembly parts because it is able to preserve as much as possible of the semantic information. However, the detection level for the bitcode generated from low-IR could be improved, for example, reconstructing typed objects from assembly, using DWARF debug information, and adding extra detection heuristics (e.g., from [23]).

Support for binaries. Even though Inception targets the analysis of source code during development, binary code may appear as a precompiled library (e.g., we have encountered this case with `libopencm3`). Since the binary is statically linked with the application, Inception can collect enough information about function prototypes, symbols, and their addresses to successfully decompile and merge the library functions used by the application. This case is handled not much differently from that of functions containing inline assembly.

Support for C/C++. Inception supports all main C types but inherits from KLEE the support for symbolic floating-point values. Regarding C++, we support the C subset. Name mangling is poorly supported by the LLVM 3.6 linker, and the syntax of some namespaces is not accepted by the Clang 3.6 front end, which is more strict than GCC 4.8. The subset that works in Inception is generally enough for embedded software and for our samples.

Manual effort. Inception reduces the manual effort required for analyzing embedded software, since it does not require any change to the original code to support assembly and peripherals. The main challenge for a user is the general problem of tuning symbolic execution. On a more practical side, Inception requires extending compilation to CLang (e.g., in presence of GCC-specific features) and to extract the memory layout of mapped memory from the datasheet. This can be at least partially automated with custom or existing tools. Moreover, compiling with CLang is worthwhile to profit from its advanced static checks.

6 Related Work

In this section we cover related work on embedded software testing and binary lifting.

Testing embedded software in an emulator and forwarding the interaction with the real hardware has been previously performed with several different approaches [32, 34, 22, 17]. Unlike Inception, Avatar [34], Prospect [32], and S2E [8] only support analysis on binary code. In [16] caching is used to reduce the memory-forwarding bottleneck. SURROGATES [17] introduces an efficient host to device debugger link. Unfortunately, the hardware is not available anymore and the software has never been publicly released. FIE [10] can perform symbolic execution of (MSP430 16-bit) source code, but it does not support assembly code and interaction with real hardware, thus requiring us to modify the application. Inception heavily relies on KLEE which uses LLVM-IR [19] bitcode generated with the CLang [33] compiler. Inception, S2E, and FIE all rely on KLEE, but only Inception's version of KLEE can handle mixed levels of abstraction and semantics. Symbolic execution is used in [4, 15] to analyze specific applications, such as BIOS or firmware in USB devices.

Lifter and its validation. The way we validated Inception's lifter is similar to the validation of the ARMv7-M formal instruction set [13] or to the testing of CPU emulators [21]. Using a machine-readable architecture specification to generate the lifter [28], or to generate test cases, would provide a higher level of assurance. However, none of the current formal descriptions for Arm processors [27, 13] support the ARMv7-M architecture. Lifters are often used for particular applications. For example, PIE [9] relies on S2E to perform static analysis, whereas FirmUSB [15] lifts binary code to perform symbolic execution. Research in lifter design is quite active. Fracture [18] tries to leverage the semantic information already present in compilers in the other direction. This approach is successful for generating bitcode for static analysis, but we found it unsuitable for generating executable LLVM bitcode and for integration with our merging step. Other approaches [31, 15, 3, 6, 14] are based on static translation, while tools such as QEMU [5] use dynamic translation, which we avoid, since integrating them with our merging approach would be complex.

7 Conclusions

In this paper we highlighted the need for handling programs as a whole in embedded systems development and testing. Like prior work, our experiments show that testing based on the source code leads to a much better bug-detection level than when working only on the binary code. These two constraints together imply that embedded programs need to be considered with both their high-level source code and their hand-written assembler code. For this purpose we compile plain C functions with LLVM toolchain into LLVM-IR and functions which include assembler into native code, which we then directly lift to LLVM-IR. Finally, we merge this code and execute it in Inception VM (a modified KLEE), which handles both abstraction levels and is able to interact with the hardware using a fast debugger. We performed extensive tests and found two new vulnerabilities and eight crashes in embedded programs, including bootloaders which were written to be included on a Mask ROM. The entire project is open-sourced to make our results easily reproducible and available at `https://github.com/Inception-framework/`.

References

[1] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In *European conference on object-oriented programming* (1996), Springer, pp. 142–166.

[2] ARM. *APCS: ARM Procedure Call Standard for the ARM Architecture*, November 2015. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf`.

[3] ARTEM DINABURG, A. R. McSema: Static Translation of X86 Instructions to LLVM, 2014.

[4] BAZHANIUK, O., LOUCAIDES, J., ROSENBAUM, L., TUTTLE, M. R., AND ZIMMER, V. Symbolic Execution for BIOS Security. *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).

[5] BELLARD, F. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[6] BOUGACH, A., AUBEY, G., COLLET, P., COUDRAY, T., SALWAN, J., AND DE LA VIEUVI, A. Dagger: Decompiling Software Through LLVM, 2013.

[7] BUYUKKURT, B., AND BAEV, I. Google groups LLVMdev RFC: Indirect Call Promotion LLVM Pass. RFC, `https://groups.google.com/forum/#!topic/llvm-dev/_1kughXhjIY`.

[8] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E Platform. *ACM Transactions on Computer Systems* (2012).

[9] COJOCAR, L., ZADDACH, J., VERDULT, R., BOS, H., FRANCILLON, A., AND BALZAROTTI, D. PIE: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, ACM, pp. 251–260.

[10] DAVIDSON, D., MOENCH, B., RISTENPART, T., AND JHA, S. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium* (2013), pp. 463–478.

[11] DIGILENT'S ZEDBOARD ZYNQ, F. Dev. board documentation. *Google Scholar*.

[12] EGNERS, A., MARSCHOLLEK, B., AND MEYER, U. Hackers in your pocket: A survey of smartphone security across platforms. Technical report RWTH Aachen , ISSN 0935–3232, May 2012.

[13] FOX, A., AND MYREEN, M. O. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving* (Berlin, Heidelberg, 2010), ITP'10, Springer-Verlag, pp. 243–258.

[14] HASABNIS, N., AND SEKAR, R. Lifting assembly to intermediate representation: A novel approach leveraging compilers. *SIGOPS Oper. Syst. Rev. 50*, 2 (Mar. 2016), 311–324.

[15] HERNANDEZ, G., FOWZE, F., TIAN, D. J., YAVUZ, T., AND BUTLER, K. R. FirmUSB: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2245–2262.

[16] KAMMERSTETTER, M., BURIAN, D., AND KASTNER, W. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)* (2016).

[17] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT* (2015).

[18] LABORATORY, C. S. D. Fracture: architecture-independent decompiler to LLVM IR, 2013.

[19] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, CGO* (2004).

[20] LI, D. X., ASHOK, R., AND HUNDT, R. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 53–61.

[21] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 261–272.

[22] MUENCH, M., NISI, D., FRANCILLON, A., AND BALZAROTTI, D. Avatar^2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)* (February 2018), BAR 18.

[23] MUENCH, M., STIJOHANN, J., KARGL, F., FRANCILLON, A., AND BALZAROTTI, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA* (San Diego, UNITED STATES, 02 2018).

[24] NXP (FREESCALE SEMICONDUCTOR). *MC1322x Advanced ZigBee^{TM}- Compliant Platform-in-Package (PiP) for the 2.4 GHz IEEE® 802.15.4 Standard*, document number: mc1322x ed. Rev. 1.3 10/2010, `https://www.nxp.com/docs/en/data-sheet/MC1322x.pdf`.

[25] OBERMAIER, J., AND TATSCHNER, S. Shedding too much light on a microcontroller's firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17). USENIX Association* (2017).

[26] PUSTOGAROV, I., RISTENPART, T., AND SHMATIKOV, V. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 757–770.

[27] REID, A. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016* (2016), pp. 161–168.

[28] REID, A. ARM releases machine readable architecture specification. Blog Post, 2017. `https://alastairreid.github.io/ARM-v8a-xml-release/`.

[29] SEMICONDUCTOR, C. Cyusb301x, cyusb201x ez-usb fx3 superspeed usb controller datasheet [r/ol]. *Cypress Semiconductor* (2016).

[30] SEREBRYANY, K. Sanitize, Fuzz, and Harden Your C ++ Code. *USENIX Security* (2015).

[31] SHEN, B.-Y., CHEN, J.-Y., HSU, W.-C., AND YANG, W. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems* (2012), ACM, pp. 51–60.

[32] SÜSSKRAUT, M., KNAUTH, T., WEIGERT, S., SCHIFFEL, U., MEINHOLD, M., AND FETZER, C. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 131–140.

[33] THE LLVM PROJECT. Clang: a C language family frontend for LLVM.

[34] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. *Proceedings of the 2014 Network and Distributed System Security Symposium* (2014).

Appendix

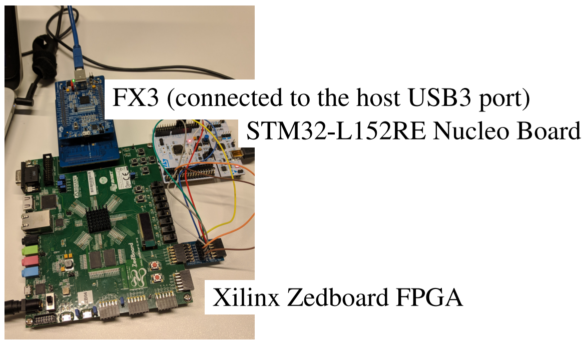

FX3 (connected to the host USB3 port)
STM32-L152RE Nucleo Board
Xilinx Zedboard FPGA

Figure 8: Hardware components of the Inception system using an STM32 demo board using an Arm Cortex-M3.

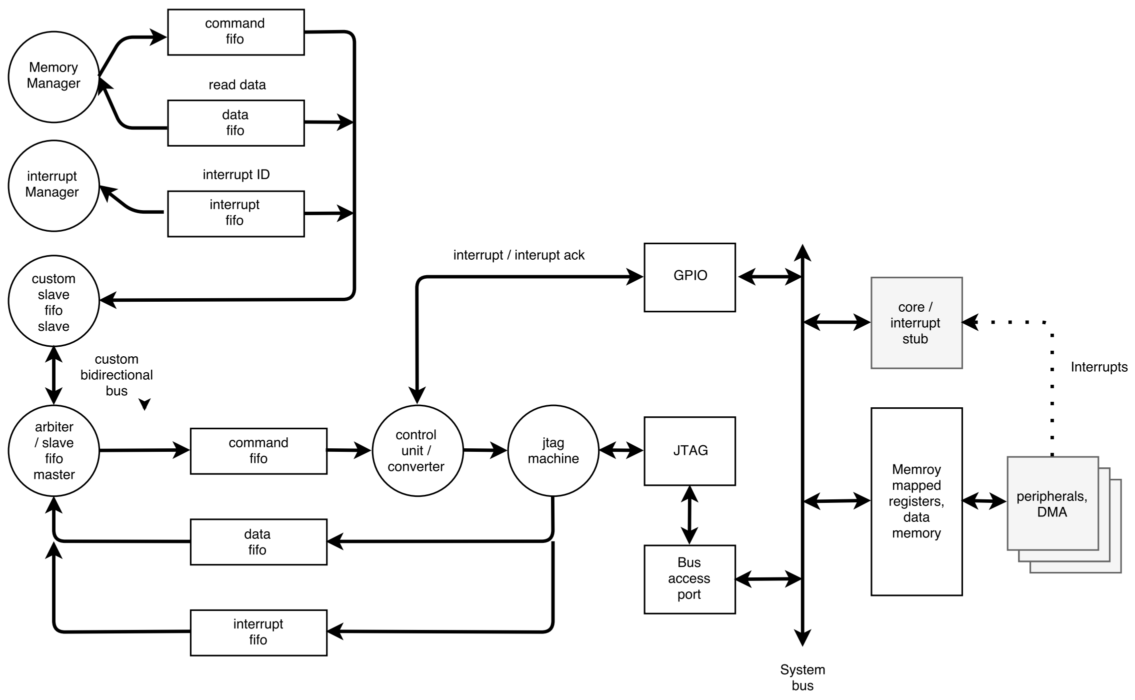


Figure 9: Overview of the forwarding process, from the managers in KLEE to the device bus, through our debugger.

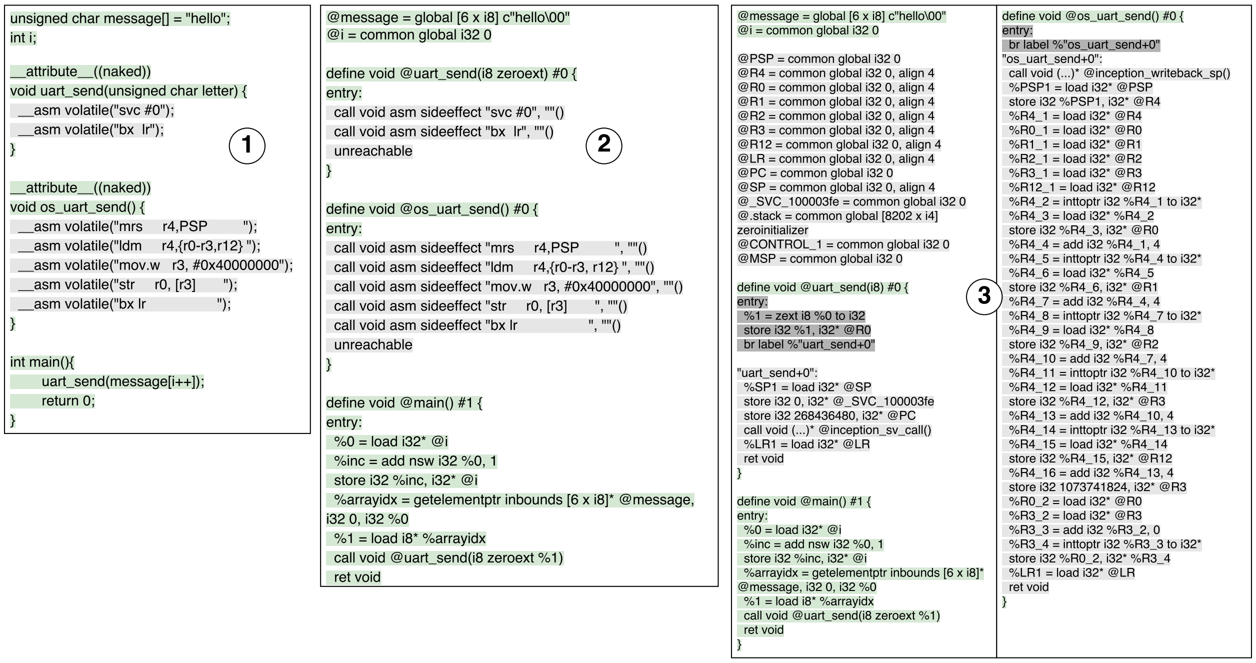
A Examples of IR level adaptation

1. **High-IR to low-IR parameters passing**.

```
define i32 @foo(i32 %a, i32 %b) #0 {
entry: // PROLOGUE BB
  store i32 %a, i32* @R0
  store i32 %b, i32* @R1
  br label %"i32x4_reti32+0"

"i32x4_reti32+0":
  ...
  //EPILOGUE
  %0 = load i32* @R0
  ret i32 %0
}
```

2. **Low-IR to high-IR parameter passing**.

```
void @high_function(){
... // High IR code

%R0_2 = load i32* @R0
%R1_1 = load i32* @R1
%R2_1 = load i32* @R2
%R3_2 = load i32* @R3
%SP15 = load i32* @SP
%SP16 = inttoptr i32 %SP15 to i32*
%SP17 = load i32* %SP16

%0 = call i32 @low_function(
      i32 %R0_2,
      i32 %R1_1,
      i32 %R2_1,
      i32 %R3_2,
      i32 %SP17)

store i32 %0, i32* @R0

... // High IR code
}

define i32 @foo(i32 %a, i32 %b,
  i32 %c, i32 %d, i32 %e) #0 {
  ... // low-IR
}
```



```
unsigned char message[] = "hello";
int i;

__attribute__((naked))
void uart_send(unsigned char letter) {
  __asm volatile("svc #0");
  __asm volatile("bx  lr");
}

__attribute__((naked))
void os_uart_send() {
  __asm volatile("mrs     r4,PSP        ");
  __asm volatile("ldm     r4,{r0-r3,r12} ");
  __asm volatile("mov.w   r3, #0x40000000");
  __asm volatile("str     r0, [r3]      ");
  __asm volatile("bx lr                 ");
}

int main(){
      uart_send(message[i++]);
      return 0;
}
```

```
@message = global [6 x i8] c"hello\00"
@i = common global i32 0

define void @uart_send(i8 zeroext) #0 {
entry:
  call void asm sideeffect "svc #0", ""()
  call void asm sideeffect "bx  lr", ""()
  unreachable
}

define void @os_uart_send() #0 {
entry:
  call void asm sideeffect "mrs     r4,PSP        ", ""()
  call void asm sideeffect "ldm     r4,{r0-r3, r12} ", ""()
  call void asm sideeffect "mov.w   r3, #0x40000000", ""()
  call void asm sideeffect "str     r0, [r3]      ", ""()
  call void asm sideeffect "bx lr                 ", ""()
  unreachable
}

define void @main() #1 {
entry:
  %0 = load i32* @i
  %inc = add nsw i32 %0, 1
  store i32 %inc, i32* @i
  %arrayidx = getelementptr inbounds [6 x i8]* @message,
i32 0, i32 %0
  %1 = load i8* %arrayidx
  call void @uart_send(i8 zeroext %1)
  ret void
```

```
@message = global [6 x i8] c"hello\00"
@i = common global i32 0

@PSP = common global i32 0
@R4 = common global i32 0, align 4
@R0 = common global i32 0, align 4
@R1 = common global i32 0, align 4
@R2 = common global i32 0, align 4
@R3 = common global i32 0, align 4
@R12 = common global i32 0, align 4
@LR = common global i32 0, align 4
@PC = common global i32 0
@SP = common global i32 0, align 4
@_SVC_100003fe = common global i32 0
@.stack = common global [8202 x i4]
zeroinitializer
@CONTROL_1 = common global i32 0
@MSP = common global i32 0

define void @uart_send(i8) #0 {
entry:
  %1 = zext i8 %0 to i32
  store i32 %1, i32* @R0
  br label %"uart_send+0"

"uart_send+0":
  %SP1 = load i32* @SP
  store i32 0, i32* @_SVC_100003fe
  store i32 268436480, i32* @PC
  call void (...)* @inception_sv_call()
  %LR1 = load i32* @LR
  ret void
}

define void @main() #1 {
entry:
  %0 = load i32* @i
  %inc = add nsw i32 %0, 1
  store i32 %inc, i32* @i
  %arrayidx = getelementptr inbounds [6 x i8]*
@message, i32 0, i32 %0
  %1 = load i8* %arrayidx
  call void @uart_send(i8 zeroext %1)
  ret void
}
```

```
define void @os_uart_send() #0 {
entry:
  br label %"os_uart_send+0"
"os_uart_send+0":
  call void (...)* @inception_writeback_sp()
  %PSP1 = load i32* @PSP
  store i32 %PSP1, i32* @R4
  %R4_1 = load i32* @R4
  %R0_1 = load i32* @R0
  %R1_1 = load i32* @R1
  %R2_1 = load i32* @R2
  %R3_1 = load i32* @R3
  %R12_1 = load i32* @R12
  %R4_2 = inttoptr i32 %R4_1 to i32*
  %R4_3 = load i32* %R4_2
  store i32 %R4_3, i32* @R0
  %R4_4 = add i32 %R4_1, 4
  %R4_5 = inttoptr i32 %R4_4 to i32*
  %R4_6 = load i32* %R4_5
  store i32 %R4_6, i32* @R1
  %R4_7 = add i32 %R4_4, 4
  %R4_8 = inttoptr i32 %R4_7 to i32*
  %R4_9 = load i32* %R4_8
  store i32 %R4_9, i32* @R2
  %R4_10 = add i32 %R4_7, 4
  %R4_11 = inttoptr i32 %R4_10 to i32*
  %R4_12 = load i32* %R4_11
  store i32 %R4_12, i32* @R3
  %R4_13 = add i32 %R4_10, 4
  %R4_14 = inttoptr i32 %R4_13 to i32*
  %R4_15 = load i32* %R4_14
  store i32 %R4_15, i32* @R12
  %R4_16 = add i32 %R4_13, 4
  store i32 1073741824, i32* @R3
  %R0_2 = load i32* @R0
  %R3_2 = load i32* @R3
  %R3_3 = add i32 %R3_2, 0
  %R3_4 = inttoptr i32 %R3_3 to i32*
  store i32 %R0_2, i32* %R3_4
  %LR1 = load i32* @LR
  ret void
}
```

Figure 10: Example program with mixed source and assembly. (1) the original C source code with inline assembly code. (2) CLang generated LLVM bitcode. (3) mixed-IR: LLVM bitcode with produced by merging lifted bitcode with CLang generated bitcode. We use the naked keyword to limit the size of the example.

Type	Board	Sample(s)	Number	Generation	Golden Model	Automated Functionality Check	Stable
Forwarding hardware	None	Test-bench	1	Manual	Python model	✓	✓
Forwarding driver	Any	IO benchmark	1	Random, manual	Property	✓	✓
Single instructions	Any	Translator-verif	50k	Random	Native regs/stack	✓	✓
Sequences, control flow	Any	Translator-verif	3k	Random	Native regs/stack	✓	✓
Feature-specific	Any	Inception-samples	13	Manual	Property	✓	✓
Simple algorithms	Any	Inception-samples	9	Manual	C version	✓	✓
Complex algorithms	STM32L152RE	Arm DSP library	4	Collected	Hardwired result	✓	✓
Complex features	Host only	mini-arm-os	3	Collected, manual	Behavior	✗	✓
Important KLEE regressions	Any	Examples	102	Collected	Property	✓	✓
Dhrystone v2.1	Host only	Performance benchmark	1	Collected	Property	✓	✓
NIST Klocwork based	Any	Vulnerable examples	40	Collected	Property	✓	✓
expat based	Any	Vulnerable examples	16	Collected	Property	✓	✓
Interrupts and multithreading	STM32L152RE	Vulnerable examples	5	Manual	Property	✓	✓
Simple demos	LPC1850DB1	Drivers for LEDs, buttons, ADC, Ethernet, Web server	5	Collected	Native behavior	✗	✓
	STM32L152RE	Drivers for LEDs, buttons, UART (ST and libopencm3)	5				✓
		Temperature via UART (libopencm3)	1				✓
	(Anonymized)	Drivers for LEDs	1				✓
Complex demos	STM32L152RE	FreeRTOS 2 threads	1	Collected	Native behavior	✗	✓
		ChibiOS	1				✗
		MbedOS	1				✗
	(Anonymized)	Bootloader	1				✓
		SDK	1				✓
		Smart Card Reader	1				✓
		MbedTLS 2.6.0	1			✓	✓
FreeRTOS and NIST Juliet	STM32L152RE	Vulnerable examples	1562	Collected			✓

Table 4: Summary of validation tests and results.