Mitigating Smart Card Fault Injection with Link-time Code Rewriting: a Feasibility Study

Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter, and Koen De Bosschere

Computer Systems Lab, Ghent University
{jmaebe,rdkeulen,brdsutte,kdb}@elis.ugent.be

Abstract. We present a feasibility study to protect smart card software against fault-injection attacks by means of binary code rewriting. We implemented a range of protection techniques in a link-time rewriter and evaluate and discuss the obtained coverage, the associated overhead and engineering effort, as well as its practical usability.

Keywords: smart card, fault injection, software protection, binary rewriting

1 Introduction

Cryptographic keys and PIN hashes are often embedded in bank smart cards. To steal that data, attackers inject faults with power glitches, clock period alterations, temperature rises, active probing of buses, or light attacks [2]. The faults cause bit flips to alter data values or program code. When this remains undetected, security barriers risk being broken: private keys can leak [1], encryption rounds are skipped [4], buffers can overflow [2], and critical checks can be skipped [8]. To protect software against such attacks, redundancy can be inserted in the code to detect occurring faults. Ideally, some tool can apply generic forms of low-overhead redundancy fully automatically to implement security policies specified in a convenient form without impeding the programmer's productivity.

Many redundancy schemes have been proposed in the past to protect software against soft errors [14] and to prevent control flow from deviating from predetermined paths [13]. However, all automated techniques that try to limit the performance overhead of the introduced redundancy is implemented in tools that do not cooperate well with other compilers.

In practice, companies rely on multiple in-house and third-party development tool chains that may change over time. To maintain interoperability with different tool chains and avoid vendor lock-in, tools that automate the implementation of security policies should therefore be separate tools that do not break existing tool chains and do not depend on their internal operation. This leaves two basic options to insert redundancy: source code rewriting and binary code rewriting.

Source-to-source rewriters essentially insert redundancy in the source code by duplicating statements. They suffer from major drawbacks. First, optimizing compilers risk undoing the protection by eliminating the redundancy they detect in the source code. Secondly, as security is a problem that concerns many abstraction and implementation layers, many security policies involve lower-level aspects that are hard to control in source code when the used compilers are black

boxes. Thirdly, source-to-source rewriters are by definition language-dependent and hence need to be redeveloped for every programming language. Finally, white and black box security testing typically takes place on the final binary code. Communication between testing teams and protection tool developers is much harder if the former are studying assembly code generated by some black-box compiler while the latter are working on source code.

Binary code rewriters do not suffer from these problems. They do suffer, however, from the fact that they have to operate on binary code that lacks symbolic information. This limits the precision and scope of many program analyses and transformations, which affects the provided level of protection and the overhead.

This paper presents a *feasibility study* of link-time binary code rewriting to protect against fault-injection attacks. We argue why such an approach is feasible. We evaluate the coverage and overhead of link-time code rewriting that implements certain security policies. The policies we examine are cookbook recipes [10] for local hardening of code against certain classes of single-instruction failures, i.e., single instructions that are skipped as the result of an injected fault.

We know of no automated fault-injection protection tools in use today for smart card software. We believe that the need to provide this protection manually is the main reason why manual assembly programming is still so common. By providing convincing arguments for the feasibility of automating this protection in a tool that does not disrupt proprietary compiler-based tool flows, we hope to contribute a significant step towards more productive smart card programming.

2 Link-time Software Protection against Fault Injection

Around the turn of the century, link-time rewriters were presented for performance optimization [12] and code compaction [6]. Today, compilers like gcc and LLVM also include so-called link-time optimizations. Those operate on the compiler's intermediate representation, however, not on binary code. These compilers therefore do not meet the practical requirements set forward in the introduction.

By contrast, Diablo (http://diablo.elis.ugent.be) is a true link-time rewriting framework [15]. Its flexibility, versatility, extensibility, retargetability, portability, and reliability have been demonstrated extensively in the past, which make it a promising candidate tool for the automated protection of binary code. In short, Diablo has been used to develop tools that rewrite code for a range of goals including optimization, compaction, (de-)obfuscation, anti-tampering, formal verification, instrumentation, GUI executable editing, and OS customization. These tools have been applied to binaries from different source languages, including C, C++, assembler, and Fortran. Several of those tools have been applied to binaries generated by different compiler generations covering 10 years of proprietary as well as open-source compilers (incl. ARM ADS, ARM RVCT, and gcc). Finally, the Diablo framework has been used to rewrite the Linux kernel for both ARM and x86. This kernel features artifacts such as code for physical and virtual address spaces, privileged instructions, manually written assembler not adhering to the conventions as specified in application binary in-

terfaces (ABIs), and a complicated, non-standard build process. For an overview of all Diablo-related research results, we refer to the Diablo website.

Together, these demonstrations show that link-time rewriters can meet the requirements discussed before: they can cooperate reliably with black box, third-party tool chains and code libraries. It is, however, not clear *a priori* that a tool like Diablo can deliver acceptable protection at acceptable overhead, with an acceptable engineering effort. Link-time tools are designed to depend only on information available in object files, such as symbol and relocation information [9]. This enables them to handle code generated by open-source as well as closed, proprietary compilers. However, this also limits their capabilities.

First, they lack high-level semantic information about the code to be rewritten. For example, no type information is available, which makes alias analysis much less precise [11]. Consequently, link-time rewriters typically need to handle memory by and large as a black box. Rather than performing register allocation globally, like compilers do, they need to find free registers locally to store temporary values. In case they cannot find them, the values have to be spilled to the stack. Applying this spilling locally in a link-time rewriter introduces considerably more overhead. Secondly, at link time indirect control flow transfers of which binary code analysis cannot resolve the targets precisely have to be modeled conservatively, i.e., over-approximated, on the basis of relocation information [5]. Through additional edges that model the over-approximation, the control flow graph (CFG) then models a superset of all possible program executions. This is safe, but leads to a loss in analysis precision. It is mainly because of these limitations that a feasibility study like this one is needed.

For this feasibility study, we implemented three first-line-of-defense cookbook protection schemes that provide local hardening against certain single-instruction failure attacks [10]. We implemented these in a tool on top of the Diablo framework for the ARM Cortex-M0 instruction set architecture (ISA), which is used in the ARM smart card SecurCore SC000 processors [16] that target future smart card applications. In these protection schemes, the program executes some "invalid state exception" code when a fault is detected. In our prototype, we opted for an infinite loop to prevent the export of any secret data.

Conditional Branch Duplication Sensitive code paths are often shielded by checks that, e.g., verify whether a correct password was entered. On smart card processors like the ARM SecurCore SC000, these checks correspond to conditional branches in the binary code. The branches are taken or not depending

```
ldr r0,[r1]
cmp r0,#5
beq .success
 <failure handling code>
.success:
 <sensitive code>
```

Fig. 1. Original code

on flags in a status register, that can either be set using an explicit comparison instruction or implicitly according to the result of an ALU operation. Attacks can focus on the input values used to perform the operation that sets the flags, on that operation itself, or on the conditional branch that depends on the flag.

To protect against attacks that make the checks ineffective by skipping one of these instructions, we duplicate the computation of the flags and the con-

ditional branch. A typical scenario targeted by this transformation is depicted in Figure 1, which is then transformed into the code of Figure 2. The shown transformation defends against all avenues of attack mentioned above: The input value is protected by duplicating the defining instruction, the flag setting is covered by duplicating the comparison, and branching based on the flag values is covered by duplicating the conditional branch.

More complex variations of the code shown in Figure 1 can occur. First, there may be multiple, different definitions of the input value(s) of the comparison in different predecessors of the comparison's basic block. In that case the duplication of the definition can be skipped, which weakens the protection because of reduced redundancy, or a significant amount of code needs to be duplicated through so-called tail duplication [11] even before the actual redundancy can be

```
  ldr r0,[r1]
  cmp r0,#5
  beq .dup1
  <failure handling code>
.dup1:
  ldr r0,[r1]
  cmp r0,#5
  beq .success
  <invalid state exception>
.Success:  <sensitive code>
```

Fig. 2. Protected code

inserted. So far, we implemented only the former in our prototype. Secondly, sometimes we cannot simply duplicate the instruction defining the value to be compared because its source operands are no longer available. This is the case, e.g., when the defining instruction is a load like `ldr r0, [r0]`. In that case, we need to find a free register to store a copy of the original source operands, and use that register in the duplicate. When no free registers are found, they are created by inserting spill code. Apart from spilling data register to the stack to free them for use in the duplicated code, it can also be necessary to spill the status register when flags are used beyond the conditional branch. This adds overhead. Finally, when the flags are set by an ALU operation that overwrites a source operand with a new value, we also needs to find a free register.

Call Graph Integrity Security analyses performed on a program's call graph are only as trustworthy as the guarantee that only modeled calls or returns can occur. By injecting bogus call or return addresses into the execution of a program, it is possible to invalidate any call graph constructed statically. Our integrity transformation works at a local level: at each individual call and return site a value is set to be checked at the intended destination. At every function entry and at every return,

```
Caller:
  ...
  mov r4,#id
  blx Callee
  ...
Callee:
  cmp r4,#id
  beq .success
  <invalid state exception>
  ...
.success:
  <sensitive code>
```

Fig. 3. ID Passing & checking

we then verify that control indeed came from one of the allowed points. This is less strong than a protection that verifies entire call chains, but it can be easily applied to call graphs with hard to analyze constructs such as recursion.

Since our transformation is applied at link time, supporting indirect function calls through function pointers or polymorphic method invocations requires extra care. Lacking type information, link-time rewriters typically cannot determine the exact targets of an indirect call. This is solved by clustering all functions that may be called indirectly (according to the symbol and relocation information

found in the object files) and treating them as a single function as far as this transformation is concerned. While this makes the protection less tight, it allows dealing statically with uncertainties introduced by dynamic program behavior.

Figure 3 shows how each check consists of two parts. Before every call a register or global variable is set to the unique identifier *id* of the callee (or cluster of callees). Next, instructions inserted in each function's prologue verify whether the set value matches its identifier (or that of its cluster). Similarly, before every return instruction a register or global variable is set to another id. This value is checked at the return points in the callers.

The process starts by partitioning the program's functions into clusters whose members can call each other indirectly or that can be called from the same indirect call site. Next, the registers free on entry and exit in all functions in a cluster are collected. If some register is always free on entry, it will be used to pass the value from the caller to the callee, otherwise the value is passed via a global variable. The same happens at the function exits.

```
str r0,[r1]
ldr r2,[r1]
cmp r0, r2
beq .ok
 <invalid state exception>
.ok:
 ...
```

Fig. 4. Store verification

Memory Store Verification The failure of a store operation at run time generally means that program state is lost. This can be addressed by checking that the store actually took place and that the correct value was written to memory. Such a transformation also introduces some resiliency to memory errors.

The proper execution of a store can be verified by loading the stored value back from memory and verifying that it matches the value that should have been stored. This happens with a comparison, as depicted in Figure 4. This transformation requires an extra free register to reload the stored value. Additionally, the status flags must be available since we insert a comparison. We do not have to duplicate the comparison in order to be safe from a single-instruction failure attack, since such an attack corrupts either the store or the comparison, but not both. Multiple attacks can be dealt with by duplicating the inserted code as many times as the number of attacks that should be handled.

Besides discussing the transformations we implemented support for, it is useful to discuss the engineering effort we invested in Diablo's core infrastructure. This demonstrates that we can build on existing infrastructure in link-time rewriters to solve technical issues of automated protection, rather than having to adapt their fundamental concepts or having to implement ad-hoc solutions.

First, inserting redundancy involves static as well as dynamic code duplication: In Figure 2, the load occurs twice in the code statically and it will be executed twice dynamically. For memory-mapped IO, this is problematic: Memory-mapped IO operations should obviously not be duplicated dynamically. So on architectures like ARM, our tool faces the problem to differentiate between normal memory operations and memory-mapped IO operations. To solve this, the software developer has to provide our tool with a list of IO register memory addresses. Diablo constant propagation analysis [11] detects instructions that access constant memory addresses. When such instructions are detected

that access IO registers, we have those instructions marked as memory-mapped IO to prevent the protecting transformations from duplicating them. This required no changes at all to Diablo's basic infrastructure. In practice, it works because memory-mapped IO is typically programmed with hard-coded constant addresses, for example through macros in the source code, for which the detection based on constant propagation works well. In theory, it is possible to write an application for which this solution will not work (and for which no fully automated solution will work, due to the undecidability of the aliasing problem).

Secondly, like all compilers, Diablo iteratively applies analyses and transformations. To simplify their implementation, most of them make some assumptions about the state of the IR. In Diablo, many analyses and transformations assume that the CFG contains no unconnected nodes. To guarantee this, an unreachable code elimination pass [11] is run before almost all analyses.

```
int some_routine(void){
    ...
    return some_value;
    check_unreachable();
}
```

Fig. 5. Unreachable code

This is problematic when a programmer wrote code that is unreachable under normal, fault-free conditions as in Figure 5, but that was added for security reasons. By default, Diablo eliminates the call to `check_unreachable`. To avoid this we adapted Diablo's basic infrastructure to keep track of the program points where it deleted unreachable code. The user of our tool can provide exception handling code, which our tool will then insert at those points before writing out the code. This provides a simple mechanism to compensate for eliminated code.
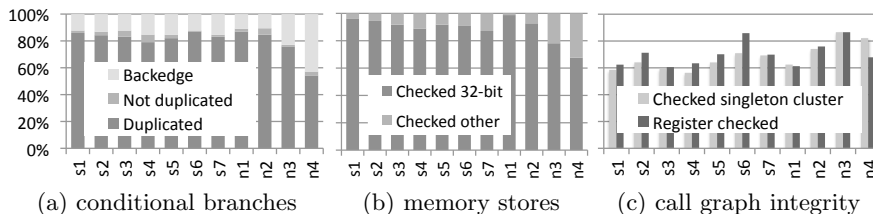
3 Experimental Evaluation

To evaluated our protections, we compiled and protected seven C MiBench [7] benchmarks (see Table 1) for a semi-hosted simulation environment (QEMU 1.0 [3]) with which we verified correctness, but on which no accurate performance can be measured. We refer to these benchmark versions as s1–s7. We used the ARM RVCT 4.1 compiler for ARM Cortex-M0 platforms with -O2. This compiler is a centerpiece of Keil

benchmark	domain	nr	size
full benchmarks for semi-hosted simulation			
basicmath_small	floating-point	s1	19480
bitcnts	integer bitcounting	s2	8204
qsort_large	3D point sorting	s3	14824
qsort_small	string sorting	s4	8012
susan	image processing	s5	32172
aes	crypto	s6	37092
sha	crypto	s7	7296
stripped-down benchmarks for native execution			
basicmath_small	floating-point	n1	12928
bitcnts	integer bitcounting	n2	3124
aes (encoder only)	crypto	n3	3760
sha	crypto	n4	1592

Table 1. Benchmarks (code size in bytes)

(http://www.keil.com/smartcards), a tool box often used in industrial smart card software development. Next, we ported four benchmarks to a USB device with an ARM Cortex-M3 with 32 kB of flash ROM and 8 kB of SRAM. This memory was too small for all benchmarks, so this limits our evaluation on real hardware to the four stripped-down benchmarks in Table 1. We will refer to these four natively executed, stripped-down programs as n1–n4. All binaries are linked statically, such that they include all needed RVCT 4.1 C library code. Whereas developers of real smart card applications would apply protections to

Fig. 6. Coverage results

only the sensitive parts of the applications, there is no notion of sensitivity in our benchmarks. They serve the purpose of estimating the potential overhead of our protections, so we made our tool apply them to the whole programs (with the exception of back edges for the conditional branch duplication).

Figure 6(a) shows the fraction of all conditional branches that get protected, together with the instructions setting the condition flags and, if available in the branch's basic block, the instruction defining the operand of that instruction. From all conditional branches, 97% on average and at least 92% per benchmark can be protected with our current implementation. As for the small fraction of branches not being duplicated, this was mainly the result of not finding the flag-setting instruction in the branch's basic block. Figure 6(b) shows that our prototype was able to insert checks for 100% of the stores. We differentiate between 32-bit and other stores because the latter require an additional instruction to mask the 32 bits of which 16 or 8 are stored. The bars in Figure 6(c) show the fractions of all call sites and return points of at which the call graph integrity is checked. The first bar depicts the number of points checked with a strong check, i.e., for which the callee is in a singleton cluster. The second bar depicts the number of points at which the identifier is passed in a register for minimal overhead, not in a global variable. A relatively large number of all calls involves clustered functions, for which we can typically not find a free register. The clustering mainly happens for C-library's use of function pointer tables.

Figure 7 depicts the code size overhead of our transformations. The combined overhead varies between 25–115%. The main reason why the combined protection overhead is bigger than the sum of the isolated overheads is that Diablo's liveness analysis becomes less precise after transformations have been applied. Another reason is that as the programs grow bigger, the corresponding ever larger branch offsets can no longer be encoded in single 16-bit Thumb2 instructions. Compared to the size overhead, the performance overhead depicted in Fig. 8 for the benchmarks for which we could conduct precise measurements is relatively small. We should remind the reader that we blindly applied the protections to the whole benchmarks, which inflates the overhead. In reality, smart card developers will likely limit them to the sensitive parts of their applications.

4   Conclusions

From previously demonstrated capabilities to reliably cooperate with black-box, third-party, industrial-strength proprietary compilers, combined with experi-
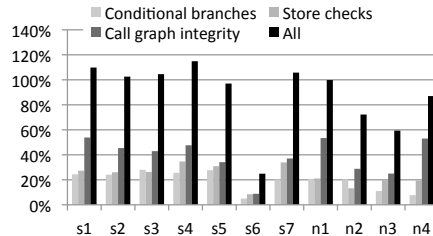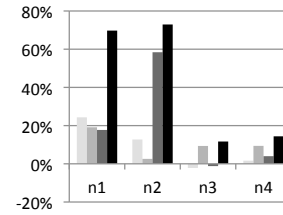
8

Fig. 7. Code size overhead

Fig. 8. Performance overhead

mental results obtained with our prototype tool, we conclude that automated link-time fault-injection protection is a realistic, promising direction.

References

1. Aumüller, C., Bier, P., Fischer, W., Hofreiter, P., Seifert, J.P.: Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In: Proc. CHES. (2002) 260–275
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Cryptology ePrint Archive, Report 2004/100 (2004)
3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proc USENIX. (2005) 41–46
4. Choukri, H., Tunstall, M.: Round reduction using faults. In: Proc. FDTC. (2005) 13–24
5. De Sutter, B., De Bus, B., De Bosschere, K.: Link-time binary rewriting techniques for program compaction. ACM Trans. Prog. Lang. and Syst. **27**(5) (2005) 882–945
6. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Prog. Lang. and Syst. **22**(2) (2000) 378–415
7. Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., Brown, R.: Mibench: A free, commercially representative embedded benchmark suite. In: Proc. IEEE WWC-4. (2001) 3–14
8. Kim, C.H., Quisquater, J.J.: Fault attacks for CRT based RSA: new attacks, new results and new countermeasures. In: Proc. WISTP. (2007) 215–228
9. Levine, J.R.: Linkers and Loaders. Morgan Kaufmann Publishers Inc. (1999)
10. Markantonakis, C., Mayes, K., Tunstall, M., Sauveron, D., Piper, F.: Smart card security. In: Computational Intelligence in Information Assurance and Security. Springer (2007) 201–233
11. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
12. Muth, R., Debray, S.K., Watterson, S., De Bosschere, K.: Alto: a link-time optimizer for the Compaq alpha. Softw. Pract. Exper. **31**(1) (2001) 67–101
13. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. IEEE Trans. Reliability **51**(1) (2002) 111–122
14. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: Proc. ACM CGO. (2005) 243–254
15. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In: Proc. ISSPIT. (2005) 7–12
16. Yiu, J.: The Definitive Guide to the ARM Cortex-M0. Newnes (2011)